

# Utility Accrual Real-Time Scheduling and Synchronization on Single and Multiprocessors: Models, Algorithms, and Tradeoffs

Hyeonjoong Cho

Dissertation submitted to the Faculty of the  
Virginia Polytechnic Institute and State University  
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy  
in  
Computer Engineering

Binoy Ravindran, Chair

James D. Arthur

Peter Athanas

Y. Thomas Hou

E. Douglas Jensen

Sandeep Shukla

Aug 24, 2006

Blacksburg, Virginia

Keywords: Real-Time Scheduling, Multiprocessors, Time/Utility Functions, Utility Accrual  
Real-Time Scheduling, Non-Blocking Synchronization, Optimality

Copyright 2006, Hyeonjoong Cho

# Utility Accrual Real-Time Scheduling and Synchronization on Single and Multiprocessors: Models, Algorithms, and Tradeoffs

Hyeonjoong Cho

(ABSTRACT)

This dissertation presents a class of utility accrual scheduling and synchronization algorithms for dynamic, single and multiprocessor real-time systems. Dynamic real-time systems operate in environments with run-time uncertainties including those on activity execution times and arrival behaviors. We consider the time/utility function (or TUF) timing model for specifying application time constraints, and the utility accrual (or UA) timeliness optimality criteria of satisfying lower bounds on accrued activity utility, and maximizing the total accrued utility.

Efficient TUF/UA scheduling algorithms exist for single processors—e.g., the Resource-constrained Utility Accrual scheduling algorithm (RUA), and the Dependent Activity Scheduling Algorithm (DASA). However, they all use lock-based synchronization. To overcome shortcomings of lock-based (e.g., serialized object access, increased run-time overhead, deadlocks), we consider non-blocking synchronization including wait-free and lock-free synchronization. We present a buffer-optimal, scheduler-independent wait-free synchronization protocol (the first such), and develop wait-free versions of RUA and DASA. We also develop their lock-free versions, and upper bound their retries under the unimodal arbitrary arrival model.

The tradeoff between wait-free, lock-free, and lock-based is fundamentally about their space and time costs. Wait-free sacrifices space efficiency in return for no additional time cost, as opposed to the blocking time of lock-based and the retry time of lock-free. We show that wait-free RUA/DASA outperform lock-based RUA/DASA when the object access times of both approaches are the same, e.g., when the shared data size is so large that the data copying process dominates the object access time of two approaches. We derive lower bounds on the maximum accrued utility that is possible with wait-free over lock-based. Further, we show that when maximum sojourn times under lock-free RUA/DASA is shorter than under lock-based, it is a necessary condition that the object access

time of lock-free is shorter than that of lock-based. We also establish the maximum increase in activity utility that is possible under lock-free and lock-based.

Multiprocessor TUF/UA scheduling has not been studied in the past. For step TUFs, periodic arrivals, and under-loads, we first present a non-quantum-based, optimal scheduling algorithm called *Largest Local Remaining Execution time-tasks First* (or LLREF) that yields the optimum total utility. We then develop another algorithm for non-step TUFs, arbitrary arrivals, and overloads, called the *global Multiprocessor Utility Accrual scheduling algorithm* (or gMUA). We show that gMUA lower bounds each activity's accrued utility, as well as the system-wide, total accrued utility.

We consider lock-based, lock-free, and wait-free synchronization under LLREF and gMUA. We derive LLREF's and gMUA's minimum-required space cost for wait-free synchronization using our space-optimal wait-free algorithm, which also applies for multiprocessors. We also develop lock-free versions of LLREF and gMUA with bounded retries. While the tradeoff between wait-free LLREF/gMUA versus lock-based LLREF/gMUA is similar to that for the single processor case, that between lock-free LLREF/gMUA and lock-based LLREF/gMUA hinges on the cost of the lock-free retry, blocking time under lock-based, and the operating system overhead.

This work was sponsored by the US Office of Naval Research under Grant N00014-00-1-0549 and The MITRE Corporation under Grant 52917.

# Acknowledgements

I believe one of the luckiest things that happened to me since I started the research in this field is that I worked with my advisor, Dr. Binoy Ravindran. In the past years, his office has been a place, where I could see a mentor who showed me a new direction every time I was stuck in research problems, where I could see a great supporter who always encouraged me to move forward, and where I could see a sincere advisor who helped me go through the trouble that might have blocked me. I would like to deeply thank Dr. Ravindran for his advising. I will remember every pleasant moment of our talk.

I would also like to thank my co-advisor, Dr. E. Douglas Jensen. It is the honor that I had many chances to receive the feedback and advice for my work from a great pioneer in the area of real-time systems. His tremendous achievements in the field give me a huge inspiration to make my idea more creative and innovative. Moreover, whenever I met him, his strong leadership and openness as a researcher used to impress me a lot. I thank Dr. Jensen again for his advising.

My other committee members offered many supportive comments as well to strengthen my work. I would like to thank Dr. James Arthur, Dr. Peter Athanas, Dr. Thomas Hou, and Dr. Sandeep Shukla for their help and time from my heart.

My M.S. degree advisor Dr. Se-young Oh in Pohang University of Science and Technology is distant from here, but his thoughtful support for his former student always feels close. I would like to specially thank Dr. Oh for his continuous help and advising.

All members of the real-time system research group also deserve my appreciation. I would like to thank Dr. Peng Li for his great help when I joined the group for the first time. Especially, I thank Dr. Haisang Wu for sharing many things about research, daily life, culture, and so on. I hope to enjoy jogging with him someday as we used to do. All other members are also great contributors for my work to be better. I would like to thank Jonathan Anderson, Edward Curley, Sherif Fahmy,

Kai Han, and Chewoo Na for their kind help and friendship.

Life in Blacksburg becomes more joyful since I am together with my friends. I would like to thank all members of our informal group, named Max5, including Jin-Suk Park, Jonghan Kim, Jong-Suk Lee, Jae-Sang Lee, Hong-Sun Lim, and Jung-Wook Suh for their friendship and valuable time we spent together. I also would like to thank all members of Korean Student Association for their assistance.

I would like to thank my family. I know my father has been praying for me every morning. I know my mother used to be sleepless in concern of her son. I also know that they are always trying not to tell me anything like those that, they are afraid, may put pressure on me. I cannot find any words enough to express my thanks to my parents as always. I also appreciate the faith and love that my fiancée's parent gives. Their unconditional and encouraging support makes me more confident every time and everywhere. My sister, my brother-in-law, and my 2-year-old niece, Yoon-joo Lim, give me their sustained love. I would like to thank my niece for the short but happy moment of walking together in hand that she suggested although she was not good at walking yet. Finally, I would like to thank my fiancée, Jina Kim. Four years ago, she helped me prepare my application to Virginia Tech. and now she will see my dissertation eventually. I appreciate her all understanding and love.

This dissertation is dedicated to all people who help me to be here.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Time Utility Functions and Utility Accrual Criteria . . . . .	2
1.2	Single Processor Scheduling and Synchronization . . . . .	3
1.3	Multiprocessor Scheduling . . . . .	8
1.4	Synchronization on Multiprocessors . . . . .	11
1.5	Summary of Contributions . . . . .	12
1.6	Organization . . . . .	16
<b>2</b>	<b>Overview of Single Processor UA Scheduling Algorithms</b>	<b>17</b>
2.1	RUA . . . . .	18
2.2	DASA . . . . .	19
2.3	Algorithm Differences . . . . .	19
<b>3</b>	<b>Synchronization on Single Processors: Lock-Based versus Non-Blocking Methods</b>	<b>21</b>
3.1	A Space-Optimal Wait-Free Protocol . . . . .	22
3.1.1	Algorithm . . . . .	22

3.1.2	Formal Comparison with Chen’s and NBW . . . . .	32
3.1.3	Numerical Evaluation Studies . . . . .	38
3.1.4	Implementation Experience . . . . .	41
3.2	UA Scheduling and Wait-Free Synchronization . . . . .	45
3.2.1	Synchronization for RUA . . . . .	45
3.2.2	Synchronization for DASA . . . . .	49
3.2.3	Implementation Experience . . . . .	50
3.3	UA Scheduling and Lock-Free Synchronization . . . . .	54
3.3.1	Wait-Free Buffer under UAM . . . . .	55
3.3.2	Bounding Retries Under UAM . . . . .	56
3.3.3	Lock-Based versus Lock-Free . . . . .	60
3.3.4	Implementation Experience . . . . .	65
<b>4</b>	<b>Multiprocessor Scheduling I: Underload Scheduling</b>	<b>71</b>
4.1	LLREF Scheduling Algorithm . . . . .	74
4.1.1	Model . . . . .	74
4.1.2	Time and Local Execution Time Plane . . . . .	74
4.1.3	Scheduling in T-L planes . . . . .	76
4.2	Algorithm Properties . . . . .	79
4.2.1	Critical Moment . . . . .	79
4.2.2	Event C . . . . .	81
4.2.3	Event B . . . . .	84

4.2.4	Optimality . . . . .	85
4.2.5	Algorithm Overhead . . . . .	88
4.3	Observations . . . . .	92
<b>5</b>	<b>Multiprocessor Scheduling II: Overload and UA Scheduling</b>	<b>93</b>
5.1	Models and Objective . . . . .	94
5.1.1	Activity Model . . . . .	94
5.1.2	Job Execution Time Demands . . . . .	95
5.1.3	Statistical Timeliness Requirement . . . . .	96
5.1.4	Scheduling Objective . . . . .	96
5.2	The gMUA Algorithm . . . . .	97
5.2.1	Bounding Accrued Utility . . . . .	97
5.2.2	Bounding Utility Accrual Probability . . . . .	97
5.2.3	Algorithm Description . . . . .	99
5.3	Algorithm Properties . . . . .	102
5.3.1	Timeliness Assurances . . . . .	102
5.3.2	Dhall Effect . . . . .	103
5.3.3	Sensitivity of Assurances . . . . .	105
5.4	Experimental Evaluation . . . . .	107
5.4.1	Performance with Constant Demand . . . . .	107
5.4.2	Performance with Statistical Demand . . . . .	110

<b>6</b>	<b>Synchronization on Multiprocessors: Lock-Based versus Non-Blocking Methods</b>	<b>113</b>
6.1	Wait-Free Buffers . . . . .	115
6.2	Lock-Free Objects . . . . .	117
6.2.1	Non-Preemptive Area . . . . .	119
6.2.2	Lock-Free Objects For gMUA . . . . .	120
6.2.3	Lock-Free Objects For LLREF . . . . .	121
6.3	Lock-Based Synchronization . . . . .	122
6.3.1	Non-Preemptive Area . . . . .	123
6.3.2	Lock-Based Synchronization for gMUA . . . . .	123
6.3.3	Lock-Based Synchronization for LLREF . . . . .	124
6.4	Tradeoffs . . . . .	125
6.4.1	Numerical Analysis under LLREF . . . . .	126
6.4.2	Numerical Analysis under gMUA . . . . .	128
<b>7</b>	<b>Past and Related Works</b>	<b>130</b>
7.1	Time/Utility Function Real-Time Scheduling . . . . .	130
7.2	Non-Blocking Synchronization . . . . .	132
7.3	Multiprocessor Real-Time Scheduling . . . . .	135
<b>8</b>	<b>Conclusions and Future Work</b>	<b>137</b>
8.1	Summary of Contributions . . . . .	138
8.2	Future Work . . . . .	140

<b>Bibliography</b>	<b>142</b>
<b>Vita</b>	<b>152</b>

# List of Figures

1.1	Example TUF Time Constraints . . . . .	3
2.1	Scheduling Objective Differences Between EDF, DASA, and RUA . . . . .	20
3.1	Typical Wait-Free Implementation . . . . .	23
3.2	Number of Buffers in Use . . . . .	24
3.3	An Inductive Approach to DSP . . . . .	26
3.4	A Worst-Case of the WFBP . . . . .	28
3.5	Decision Procedure . . . . .	35
3.6	Buffer Sizes Under Increasing Interferences With Normal Distribution for $N_i^{max}$ . . . . .	38
3.7	Buffer Sizes Under Different $N_i^{max}$ Distributions with 40 Readers . . . . .	39
3.8	Buffer Sizes With 2 Reader Groups Under Varying Reader Ratio, for 20, 30, and 40 Readers . . . . .	40
3.9	Buffer Sizes With 3 Reader Groups Under Varying Reader Ratio, for 20, 30, and 40 Readers . . . . .	41
3.10	ACET of Read/Write in SHaRK RTOS . . . . .	43
3.11	WCET of Read/Write in SHaRK RTOS . . . . .	43

3.12	Buffer Sizes . . . . .	45
3.13	Performance of Lock-Based and Wait-Free DASA Under Increasing Number of Shared Objects . . . . .	51
3.14	Performance of Lock-Based and Wait-Free DASA Under Increasing Number of Readers . . . . .	52
3.15	Performance of Lock-Based and Wait-Free RUA Under Increasing Number of Shared Objects . . . . .	53
3.16	Performance of Lock-Based and Wait-Free RUA Under Increasing Number of Readers . . . . .	53
3.17	Three Dimensions of Task Model . . . . .	56
3.18	Mutual Preemption Under RUA . . . . .	58
3.19	Interferences under UAM . . . . .	59
3.20	Object Access Time . . . . .	66
3.21	Critical Time Miss Load . . . . .	67
3.22	AUR/CMR During Underload, Step TUFs . . . . .	68
3.23	AUR/CMR During Underload, Heterogeneous TUFs . . . . .	68
3.24	AUR/CMR During Overload, Step TUFs . . . . .	69
3.25	AUR/CMR During Overload, Heterogeneous TUFs . . . . .	69
3.26	AUR/CMR During Increasing Readers, Heterogeneous TUFs . . . . .	70
4.1	A Task Set that Global EDF Cannot Schedule On Two Processors . . . . .	72
4.2	Scheduling Objective Differences Between global EDF and LLREF . . . . .	73
4.3	Fluid Schedule versus a Practical Schedule . . . . .	75

4.4	T-L Planes . . . . .	76
4.5	$k^{th}$ T-L Plane . . . . .	77
4.6	Example of Token Flow . . . . .	79
4.7	Critical Moment . . . . .	80
4.8	Event C . . . . .	82
4.9	Linear Equation for Event C . . . . .	83
4.10	Event B . . . . .	84
4.11	Linear Equation for Event B . . . . .	86
4.12	Token Flow when $N \leq M$ . . . . .	86
4.13	Scheduler Invocation Frequency with 4 Tasks . . . . .	90
4.14	Scheduler Invocation Frequency with 8 Tasks . . . . .	91
5.1	Scheduling Objective Differences Between gMUA and Global EDF . . . . .	93
5.2	Transformation Array and gMUA . . . . .	98
5.3	Performance Under Constant Demand, Step TUFs . . . . .	108
5.4	Performance Under Constant Demand, Heterogeneous TUFs . . . . .	109
5.5	Performance Under Statistical Demand, Step TUFs . . . . .	110
5.6	Performance Under Statistical Demand, Heterogenous TUFs . . . . .	111
6.1	Reader and Writer Execution Time Line . . . . .	116
6.2	An Example of Continuous Retry . . . . .	118
6.3	LLREF-scheduled Synchronization Overhead for NPA-assisted Mechanism with Varying Tasks' Execution Time Costs . . . . .	127

6.4	LLREF-scheduled Synchronization Overhead for NPA-assisted Mechanism with Varying Number of Shared Objects . . . . .	127
6.5	gMUA-scheduled Synchronization Overhead for NPA-assisted Mechanism with Varying Tasks' Execution Time Costs . . . . .	128
6.6	gMUA-scheduled Synchronization Overhead for NPA-assisted Mechanism with Varying Number of Shared Objects . . . . .	129

# List of Tables

1.1	Coverage of Dissertation's LLREF and gMUA Scheduling Algorithms . . . . .	13
1.2	Coverage of Dissertation's Synchronization Methods . . . . .	14
3.1	Task Set . . . . .	36
3.2	Asymptotical Time Complexities . . . . .	37
3.3	Decision procedure on 16 Fast and 4 Slow Readers . . . . .	44
3.4	Experimental Parameters for Tasks . . . . .	51
4.1	Task Parameters (4 Task Set) . . . . .	90
4.2	Task Parameters (8 Task Set) . . . . .	91
5.1	Task Settings . . . . .	110

# Chapter 1

## Introduction

This dissertation focuses on utility accrual real-time scheduling and synchronization in dynamic, single and multiprocessor systems. Dynamic real-time systems are emerging in many application domains including robotics, space, defense, consumer electronics, and financial markets. Such systems are fundamentally distinguished by the fact that they operate in environments with dynamically uncertain properties. These uncertainties include transient and sustained resource overloads due to context-dependent activity execution times and arbitrary activity arrival patterns. Nevertheless, such systems desire the strongest possible assurances on activity timeliness behavior.

Another important distinguishing feature of such dynamic systems is that they also have activities that are subject to non-deadline time constraints, such as those where the utility attained for activity completion varies with completion time. This is in contrast to classical deadlines, where a positive utility is accrued for completing the activity anytime before the deadline, after which zero, or infinitively negative utility is accrued. Yet another important distinguishing feature of these systems is their relatively long execution time magnitudes, compared to conventional real-time subsystems—e.g., in the order of milliseconds to minutes, or seconds to minutes.

Like most real-time systems, application activities of dynamic systems also share physical (e.g., CPU, I/O) and logical (e.g., locks) resources under mutual exclusion constraints, and they de-

sire atomic behavior while concurrently and mutually exclusively accessing those resources. This causes resource contention, and their resolution affect system's timeliness behavior. Furthermore, multiprocessor architectures (e.g., Symmetric Multi-Processors or SMPs, Single Chip Heterogeneous Multiprocessors or SCHMs) are also becoming increasingly attractive for dynamic real-time systems because of their continuously decreasing cost (like for most real-time and non real-time systems).

Example systems that motivate our work (from the defense domain) include sensor applications such as Active Electronically Steerable Array (AESA) radars [35], combat platform applications such as surveillance aircraft [28], and large-scale enterprise applications such as network-centric warfare [16].

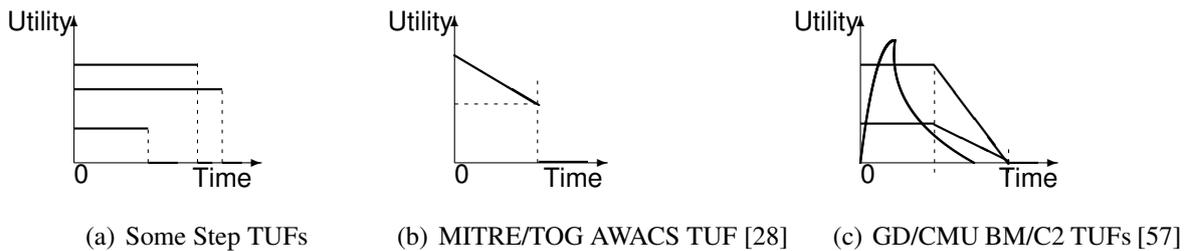
## 1.1 Time Utility Functions and Utility Accrual Criteria

When resource overloads occur, meeting deadlines of all activities is impossible as the demand exceeds the supply. The urgency of an activity is typically orthogonal to the relative importance of the activity — e.g., the most urgent activity can be the least important, and vice versa; the most urgent can be the most important, and vice versa. Hence when overloads occur, completing the most important activities irrespective of activity urgency is often desirable. Thus, a clear distinction has to be made between urgency and importance, during overloads. (During under-loads, such a distinction need not be made, because deadline-based scheduling algorithms such as *Earliest Deadline First* [45] (or EDF) are optimal on one processor — i.e., they can satisfy all deadlines.)

Deadlines by themselves cannot express both urgency and importance. Thus, we consider the timeliness model of time/utility functions (or TUFs) [47] for specifying application time constraints. A TUF is a generalization of the classical deadline constraint, and specifies the utility to the system resulting from the completion of an application activity as an application- or situation-specific function of that activity's completion time. Figure 1.1 shows examples. A TUF's utility values are derived from application-level quality of service metrics (e.g., track quality or track importance in

a radar-based tracking system [28]).

Note that a TUF directly decouples importance and urgency — i.e., urgency is measured as a deadline on the function’s X-axis, and importance is denoted by utility on the function’s Y-axis. A classical deadline is unit-valued, since importance is not considered. Downward step TUFs (Figure 1.1(a)) are a generalization of classical deadlines.



**Figure 1.1:** Example TUF Time Constraints

When activity time constraints are expressed with TUFs, the timeliness optimality criteria are typically based on accrued activity utility — e.g., maximizing sum of the activities’ attained utilities or assuring satisfaction of lower bounds on activities’ maximal utilities. Such criteria are called utility accrual (or UA) criteria, and algorithms that optimize UA criteria are called UA algorithms.

UA algorithms that maximize total utility under step TUFs (see algorithms in [64]) default to EDF during under-loads, since EDF satisfies all deadlines during under-loads (on one processor). Consequently, they obtain the maximum total utility during under-loads. During overloads, they favor more important activities (since more utility can be attained from them), irrespective of urgency. Thus, deadline scheduling’s optimal timeliness behavior is a special-case of UA scheduling.

## 1.2 Single Processor Scheduling and Synchronization

The dissertation presents a class of UA scheduling and synchronization algorithms. First, the dissertation focuses on the single processor problem space, for which UA scheduling has been well studied and several algorithms have been developed (see algorithms in [64]). However, most

of these algorithms rely on lock-based synchronization for concurrent, mutually exclusive access to shared resources.

Further, many lock-based protocols typically incur additional run-time overhead due to scheduler activations that occur when activities request locked objects [64, 65]. Lock-based synchronization has several disadvantages such as

- (1) serialized access to shared objects, resulting in reduced concurrency and thus reduced resource utilization;
- (2) increased run-time overhead due to the increased context switching between activities blocked on shared objects and their lock holders. This occurs when the blocked activity has to be preempted, the lock holder has to be executed until the lock is released, and the blocked activity's execution has to be resumed;
- (3) possibility of deadlocks that can occur when lock holders crash, causing indefinite starvation to blocked activities;
- (4) need for a-priori knowledge of the ceilings of locks (e.g., Priority Ceiling Protocol [65]), which may be difficult to obtain for dynamic applications; and
- (5) reduced flexibility due to the need to update OS data structures and recompile the OS to accommodate changes to lock ceilings.

To overcome these difficulties, we consider *wait-free synchronization* for the single-writer/multiple-reader (or SWMR) problem — a very common synchronization problem that occurs in most embedded real-time systems. Wait-free synchronization is an example *non-blocking* synchronization mechanism, where *all* activities complete their access to shared objects in finite number of steps, regardless of the number of interferences<sup>1</sup> suffered by them from other activities [46]. Consequently, no activity starves.

---

<sup>1</sup>An interference occurs when an operation on an object does not complete and another operation on the object begins. An example way by which interference occurs is preemption, i.e., an operation on an object is preempted by another operation on it as the scheduler deems the latter operation to have greater execution eligibility.

An example way to achieve wait-free synchronization is by using multiple buffers for the shared object and by carefully scheduling the execution of readers and writers to avoid interference [46].<sup>2</sup> To bound the number of buffers that are needed, the key idea is to use as many buffers as the maximum number of times the readers can be preempted by the writer. The maximum number of preemptions of a reader bounds the number of times the writer can update the object while the reader is reading. Thus, by using as many buffers as the worst-case number of preemptions of readers, the readers and the writer can continuously read and write in different buffers, respectively, and thereby avoid interference. Wait-free synchronization may incur additional time costs for their intrinsic control mechanisms, or may incur additional space cost due to the use of multiple buffers.

The dissertation presents an analytical solution to the problem of determining the minimum number of buffers that is required to ensure the safety and orderliness of wait-free synchronization in SWMR, without considering any scheduler knowledge. We call this problem, the Wait-Free Buffer size decision Problem (or WFBP). We prove that our solution to the WFBP requires lesser or equal number of buffers than what is needed by the previously best wait-free protocols for this problem, including Chen’s protocol [19], Improved Chen’s protocol [46], and NBW protocol [48]. In particular, we prove that our solution to the WFBP subsumes the number of buffers required by Chen’s protocol and the NBW protocol as special cases. Further, we analytically identify the conditions under which our protocol needs less (and equal) number of buffers than other protocols.

We also present a wait-free protocol that utilizes the minimum buffer requirement determined by our solution. Our implementation of the protocol in the SHaRK RTOS and experimental studies using that implementation validates our analytical results. Among the class of wait-free protocols that consider a-priori knowledge of interferences and no scheduler knowledge, our optimal space lower bound is the first such bound that is analytically established.

Armed with this result, we consider UA scheduling with wait-free synchronization. The motivation to do so (instead of lock-based synchronization) is to reap the advantages of wait-free synchronization (e.g., reduced object access time, greater concurrency, reduced run-time overhead,

---

<sup>2</sup>Another strategy is to use cooperative scheduling [5].

fault-tolerance) toward better optimization of UA criteria. In particular, we hypothesize that the reduced shared object access time under wait-free synchronization will result in increased activity attained utility, but at the expense of additional buffer cost. Thus, the goal of considering UA scheduling with wait-free synchronization is to establish the fundamental tradeoffs between wait-free and lock-based object sharing for UA scheduling.

The dissertation considers the UA optimality criteria of maximizing the sum of the activities' attained utilities, while yielding optimal total utility for step TUFs during under-loads, and ensuring the integrity of shared data objects under concurrent access. We consider two lock-based UA algorithms that match these exact UA criteria: (1) the Resource-constrained Utility Accrual (or RUA) scheduling algorithm [80] and (2) the Dependent Activity Scheduling Algorithm (or DASA) [27]. We develop wait-free versions of RUA and DASA using our space-optimal protocol, and analytically compares RUA's and DASA's wait-free and lock-based versions. We derive lower bounds on the maximum possible accrued utility with wait-free over their lock-based counterparts, while incurring the minimum possible additional space costs. Further, our measurements from a POSIX RTOS implementation reveal that during under-loads, wait-free algorithms yield optimal total utility for step TUFs and significantly higher utility (than lock-based) for non-step TUFs.

The tradeoff between wait-free RUA/DASA and lock-based RUA/DASA is fundamentally about their space and time costs. The wait-free approach costs space — even though the space cost is minimum and optimal for wait-free operations — in return for no additional time cost, i.e., no blocking time as opposed to the lock-based approach. Specifically, we show that “no blocking time” does not always mean a better timeliness performance for wait-free over lock-based. Comparing both approaches exclusively in terms of the time cost (i.e., task sojourn times<sup>3</sup>), we develop the condition for wait-free RUA/DASA to outperform lock-based RUA/DASA. Wait-free outperform lock-based when the object access times of both approaches are the same, e.g., when the shared data size is as large as the data copying process dominate the object access time of two approaches. We believe that these cases are common.

---

<sup>3</sup>The sojourn time is defined as the period from a task's release to its completion.

Wait-free synchronization requires an upper bound on the number of jobs which can arrive at runtime (the job upper bound is needed to upper bound the number of buffers needed for wait-free synchronization). Consequently, this technique does not scale for the dissertation's motivating dynamic application systems that have arbitrary number of job arrivals, and multiple-writer/multiple-reader (or MWMR) scenarios, resulting in unbounded number of jobs. Thus, we consider *lock-free* synchronization for this problem space.

Lock-free synchronization, in contrast to wait-free, guarantees that *some* object operations will complete in a finite number of steps. It implies that some operations may starve. An example way to achieve lock-free synchronization is through *retries* i.e., readers are allowed to concurrently read while the writer is writing (without acquiring locks). But the readers check whether their reading was interfered by the writer. If so, they read again. Thus, a reader continuously reads, checks, and retries until its read becomes successful. Since a reader's worst-case number of retries depends upon the worst-case number of times the reader is preempted by the writer, the additional execution-time overhead incurred for the retries is bounded by the number of preemptions. Inevitably, lock-free synchronization incurs additional time costs due to their retries, which is antagonistic to timeliness optimization in real-time systems.

We consider UA scheduling with lock-free synchronization. We develop the lock-free version of RUA and DASA, and derive their retry upper bound under the *unimodal arbitrary arrival model* (or UAM). UAM specifies the maximum number of activity arrivals that can occur during any time interval, while allowing arbitrary arrival frequency (and thus unbounded number of jobs) within the time interval. Consequently, the model subsumes most traditional arrival models (e.g., frame-based, periodic, sporadic) as special cases. Since lock-free sharing incurs additional time overhead due to the retries (than lock-based), we establish the conditions under which activity sojourn times are shorter under lock-free RUA/DASA than under lock-based RUA/DASA, for the UAM. When sojourn times under lock-free is shorter than that under lock-based, it is a necessary condition that the object access time of lock-free is shorter than lock-based. From this result, we establish the maximum increase in activity utility that is possible under lock-free RUA/DASA over lock-based. Further, measurements from our implementation on a POSIX RTOS reveal that lock-free RUA

yields significant increase in accrued utility, by as much as 65%, and deadline satisfactions, by as much as 80%, over lock-based.

With this result, the dissertation is thus able to provide completeness — for the first time — to the single-processor solution space on UA scheduling and (lock-based and non-blocking) synchronization.

### 1.3 Multiprocessor Scheduling

The dissertation now considers the multiprocessor problem space on UA scheduling and synchronization. Multiprocessor architecture (e.g., Symmetric Multi-Processors or SMPs, Single Chip Heterogeneous Multiprocessors or SCHMs) are becoming more attractive for embedded systems primarily because major processor manufacturers (Intel, AMD) are making them decreasingly expensive. This makes such architectures very desirable for embedded system applications with high computational workloads, where additional, cost-effective processing capacity is often needed. Responding to this trend, RTOS vendors are increasingly providing multiprocessor platform support—e.g., QNX Neutrino is now available for a variety of SMP chips [62]. But this exposes the critical need for real-time scheduling for multiprocessors—a comparatively undeveloped area of real-time scheduling which has recently received significant research attention, but is not yet well supported by the RTOS products. Consequently, the impact of cost-effective multiprocessor platforms for embedded systems remains nascent.

One unique aspect of multiprocessor real-time scheduling is the degree of run-time migration that is allowed for job instances of a task across processors (at scheduling events). Example migration models include: (1) *full migration*, where jobs are allowed to arbitrarily migrate across processors during their execution. This usually implies a global scheduling strategy, where a single shared scheduling queue is maintained for all processors and a processor-wide scheduling decision is made by a single (global) scheduling algorithm; (2) *no migration*, where tasks are statically (off-line) partitioned and allocated to processors. At run-time, job instances of tasks are scheduled

on their respective processors by processors' local scheduling algorithm, like single processor scheduling; and (3) *restricted migration*, where some form of migration is allowed — e.g., at job boundaries.

The partitioned scheduling paradigm has several advantages over the global approach. First, once tasks are allocated to processors, the multiprocessor real-time scheduling problem becomes a set of single processor real-time scheduling problems, one for each processor, which has been well-studied and for which optimal algorithms exist. Second, not migrating tasks at run-time means reduced run-time overheads as opposed to migrating tasks that may suffer cache misses on the newly assigned processor. If the task set is fixed and known a-priori, the partitioned approach provides appropriate solutions [13].

The global scheduling paradigm also has advantages over the partitioned approach. First, the partitioned approach cannot produce optimal real-time schedules — one that meets all tasks deadlines when tasks utilization demand does not exceed the total processor capacity — for periodic tasks sets [69], since the partitioning problem is analogous to the bin-packing problem which is known to be NP-hard in the strong sense. Second, if tasks can join and leave the system at run-time, then it may be necessary to reallocate tasks to processors unlike in the partitioned approach [13]. Third, in some embedded processor architectures with no cache and simpler structures, the overhead of migration has a lower impact on the performance [13]. Finally, global scheduling can theoretically contribute to an increased understanding of the properties and behaviors of real-time scheduling algorithms for multiprocessors. (See [44] for detailed discussion on this.) For these reasons, we focus on global scheduling approaches in this dissertation. We also focus on homogeneous multiprocessors like SMPs.

The global EDF for multiprocessors, one of global scheduling approaches, is subject to the “Dhall effect” [33], where a task set with total processor utilization arbitrarily close to 1.0 cannot be scheduled satisfying all deadlines even with infinite processor resources. To overcome this, some researchers have studied global EDF's behavior under restricted individual task utilization demands, and others have proposed several variants of EDF, e.g., *Earliest Deadline until Zero Laxity*

(or EDZL), which allows another scheduling event to split tasks [55]. However, they do not provide the optimality for multiprocessors.

The Pfair class of algorithms [11], another global scheduling approach, have been shown to be theoretically optimal. However, Pfair algorithms incur significant run-time overhead due to their quantum-based scheduling approach [31, 69]: under Pfair, tasks are decomposed into several small uniform segments, which are then scheduled, causing frequent scheduling and migration.

In the first place, the dissertation considers the classical hard real-time special case of UA scheduling: Scheduling on multiprocessors to meet all deadlines under the periodic arrival model.<sup>4</sup> For this problem, we present an optimal scheduling algorithm called LLREF that can meet all deadlines as long as the total utilization demand does not exceed the total processing capacity of all processors. We design LLREF using a novel abstraction for reasoning about task execution behavior on multiprocessors: *the Time and Local Execution Time Domain Plane (or T-L plane)*. LLREF is based on the fluid scheduling model and the fairness notion [11], and uses the T-L plane to describe fluid schedules without using time quanta, unlike the time quanta-based Pfair algorithm [11], which is the only known optimal algorithm for the same problem.<sup>5</sup> We show that scheduling for multiprocessors can be viewed as repeatedly occurring T-L planes, and feasibly scheduling on a single T-L plane results in the optimal schedule. We analytically establish the optimality of LLREF. Further, we establish that the algorithm has bounded overhead, and this bound is independent of time quanta (unlike Pfair). Our simulation results validate our analytical results.

We build upon LLREF with a UA scheduling algorithm that maximizes total accrued utility called the *global Multiprocessor Utility Accrual scheduling algorithm* (or gMUA) that considers dynamic models. gMUA considers a stochastic execution-time model, where activity execution demand is stochastically expressed. Further, gMUA allows activities to repeatedly arrive with a known min-

---

<sup>4</sup>The hard real-time objective is a special case of the UA objective. It can be formulated as a UA objective that requires that the product of the utility accrued by all activities must be one, where each activity is subject to a downward step TUF with a maximum utility of one and minimum utility of zero.

<sup>5</sup>Though the Pfair algorithm is optimal, the algorithm is impractical due to its significant run-time overhead, which is caused by the quantum-based scheduling approach.

imum inter-arrival time, which can be violated at run-time. We show that gMUA (1) achieves optimal total utility for the special case of step TUFs and when utilization demand does not exceed global EDF's feasible utilization bound, (2) probabilistically lower bounds the utility accrued by each activity, and (3) lower bounds the system-wide total accrued utility. We also show that gMUA's timing assurances have bounded sensitivity to variations in execution time demand estimates, and that the algorithm is robust against the Dhall effect.

## 1.4 Synchronization on Multiprocessors

Compared to the research efforts on single and multiprocessor scheduling, synchronization for multiprocessor scheduling has been less studied. For example, the synchronization under global EDF was considered only recently in [32]. We consider several resource sharing methods under multiprocessor scheduling, including lock-based, lock-free, and wait-free synchronization for both LLREF and gMUA algorithms.

In doing so, we follow the same approach as in single processor scheduling. First, we consider bounded number of jobs, the SWMR scenario, and wait-free synchronization, and analytically derive LLREF's and gMUA's minimum-required space costs using our optimal wait-free synchronization algorithm that was previously discussed. This wait-free synchronization algorithm also applies for multiprocessors.

As mentioned before, the tradeoff between wait-free and the other approaches including lock-based and lock-free is fundamentally about their space and time costs. Wait-free sacrifices space efficiency in return for no additional time cost, as opposed to the blocking time of lock-based and the retry time of lock-free. As in the single processor case, we show that the wait-free approach is beneficial when the object access times of the approaches are similar, or when the shared data size is as large as the blocking time of lock-based or the retry time of lock-free.

We then consider unbounded number of jobs and the MWMR scenario, and lock-free synchronization. We bound the retries needed for LLREF and gMUA under lock-free, and compute their

feasible utilization demand bound to satisfy all deadlines. Finally, we consider lock-based synchronization for both scheduling algorithms and derive their feasible utilization demand bound.

Tradeoff between lock-free LLREF/gMUA and lock-based LLREF/gMUA hinges on the cost of the lock-free retry, blocking time under lock-based, and the operating system overhead. Under LLREF and gMUA, the lock-free and lock-based approaches that we consider are both assisted by *non-preemptive area* (NPA), i.e., the operations of both approaches are allowed to execute inside the NPA and thus, no preemption occur during the NPA, which helps their operations bounded in time. Thus, the tradeoff also depends on the lengths of their NPAs, which are determined based on the cost of their object access times. Tradeoff between lock-free LLREF/gMUA and lock-based LLREF/gMUA hinges primarily on the object access time of both approaches. We analytically compare lock-free and lock-based approaches under LLREF and gMUA in terms of their feasible utilization demand bound to meet all time constraints.

With these results, the dissertation provides a complete, fundamental foundation to the multiprocessor solution space on UA scheduling and (lock-based and non-blocking) synchronization.

## 1.5 Summary of Contributions

We now summarize the dissertation's research contributions. Table 1.1 highlights the dissertation's contributions in the real-time scheduling problem space, contrasting them with past work, using a *three-dimensional* matrix. Each matrix row of the table represents a class of TUF shape: Step-shaped TUF and non-step shaped TUF (e.g., linear, parabolic). Each matrix column of the table represents a processor class: Single processor and multiprocessors. Each matrix element of the table (identified by a row and a column position) is partitioned into underload and overload models — the bottom left triangle of the element represents the underload model and the top right triangle represents the overload model. By the underload model, we mean the load model where the total

utilization demand of the application tasks which must be satisfied by the TUF's critical time<sup>6</sup> does not exceed the total processor capacity. In contrast, in the overload model, that total utilization demand exceeds the total processor capacity.

Algorithms developed in the past and those developed in this dissertation are now catalogued in the appropriate matrix elements (the dissertation algorithms are shown in bold). Note that this cataloguing is mutually exclusive — overload scheduling algorithms also apply for under-loads, but not vice versa; non-step TUF algorithms also apply for step TUFs, but not vice versa. Thus, we catalogue such algorithms as belonging to that category to which they uniquely apply.

**Table 1.1:** Coverage of Dissertation's LLREF and gMUA Scheduling Algorithms

	Overload	Single Processor	Multiprocessors
Underload			
Step TUFs		$D^{over}$ , DASA-ND RM, EDF, LLF, etc.	global EDF, Pfair, <b>LLREF</b> <b>gMUA</b>
Non-Step TUFs		LBESA, GUS, RUA	<b>gMUA</b>

A significant number of past research efforts have focused on real-time scheduling for deadline time constraints, under-loads, and single processors. Example algorithms in this category include *Rate-Monotonic* (or RM), EDF, and *Least-Laxity First* [59] (or LLF). In Table 1.1, we include such algorithms as algorithms for step TUFs, although strictly speaking, deadlines are a special case of step TUFs.  $D^{over}$  [49] is an optimal overload scheduling algorithm, as it is able to obtain the maximum possible competitive ratio. The *DASA without dependency* (or DASA-ND) [27] algorithm also considers step TUFs and overloads. Non-step TUFs were considered for the first time by LBESA [56], and subsequently by GUS [54] and RUA [80].

The global EDF and Pfair algorithms consider deadline time constraints and under-loads on multiprocessors. As previously discussed, both algorithms have serious disadvantages. Global EDF is not optimal for under-loads — i.e., it cannot ensure that all tasks meet their deadlines or TUF

<sup>6</sup>Critical time is the time at which the task TUF has zero utility—i.e., the "deadline" time (all TUFs are assumed to have such a time).

critical times, even when the total utilization demand is less than the capacity of all processors. The Pfair algorithm, on the other hand, can do so, but suffers from significant run-time overhead due to its time quantum-based approach. In contrast, the dissertation’s LLREF algorithm, is not time quantum-based, has lesser overhead, and can meet all deadlines or critical times as long as the total utilization demand does not exceed the capacity of all processors.

LLREF is restricted to under-loads and step TUFs. The dissertation’s gMUA algorithm targets the problem space of overloads and non step TUFs. However, gMUA is not optimal for under-loads, unlike LLREF.

**Table 1.2:** Coverage of Dissertation’s Synchronization Methods

	Overload	<i>Single Processor</i>	<i>Multiprocessors</i>
Underload			
<i>Lock-Based</i>		RUA+LB DASA+LB	<b>gMUA+LB</b>  <b>LLREF+LB</b>
<i>Non-Blocking</i>		<b>RUA+LF</b> <b>DASA+LF</b>	<b>gMUA+LF</b> <b>gMUA+WF</b> <b>LLREF+WF,LF</b>

Table 1.2 highlights the dissertation’s contributions in the real-time synchronization problem space, contrasting them with past work, using a similar three-dimensional matrix. Similar to Table 1.1, each matrix column here also represents a processor class: Single processor and multiprocessors. Each matrix row represents the type of synchronization mechanism: lock-based (LB) and non-blocking (e.g., lock-free (LF), wait-free (WF) synchronization). Each matrix element is partitioned into underload and overload models — the bottom left triangle of the element represents underload model and the top right triangle represents overload model.

Algorithms developed in the past and those developed in this dissertation are now catalogued in the appropriate matrix elements, with the dissertation algorithms shown in bold. As before, this cataloguing is mutually exclusive — algorithms for unbounded number of jobs also apply for bounded number of jobs, but not vice versa. Thus, we catalogue such algorithms as belonging to that category to which they uniquely apply.

Significant number of research efforts have focused on synchronization under deadline time constraints on single processors. Examples include: (1) LB synchronization—e.g., Priority Inheritance Protocol (PIP) and Priority Ceiling Protocol (PCP) [65], Stack Resource Policy (SRP) [9]; and (2) non-blocking synchronization including (a) WF synchronization—e.g., NBW protocol [48], Chen’s protocol [19], [5, 46], and (b) LF synchronization—e.g., [4, 6, 58, 76]. PIP and PCP has been used to bound blocking times under RM in [65], and SRP has been used to bound blocking times under RM and EDF in [9]. In [23], bounds on buffers are established for scheduler-independent WF synchronization and it is proved to be optimal in space costs. In [6], bounds for retries are established for LF synchronization under RM and EDF. The bounds for retries are used to establish the conditions under which all deadlines (or TUF critical times) can be met, both for bounded and unbounded number of jobs. However, these works do not consider overloads (caused by unbounded number of jobs and execution overruns) and non-step TUFs.

DASA [27] and RUA [80] consider step and non-step TUFs, respectively, and overloads. However, they do not consider non-blocking synchronization; so the tradeoffs with respect to LB synchronization are not known.

Synchronization on multiprocessors under deadline time constraints has also been studied, though less extensively. Examples include: (1) LB synchronization—e.g., an extension of PCP for partitioned RM [63], queue-based spin locks for EDF [32] and several locking synchronization protocols Pfair [42], and (2) LF synchronization—e.g., for EDF and Pfair [32] [43].

The dissertation algorithms target this unexplored problem space of non step TUFs, overloads, and non-blocking synchronization. The algorithms include the space optimal WF synchronization algorithm, and WF-versions of DASA, RUA, LLREF, and gMUA using that algorithm. WF synchronization is possible only with bounded number of jobs and thus is limited to under-loads. Thus, LB and LF synchronization are studied for the case of unbounded number of jobs and overloads. The dissertation presents LB versions of RUA, DASA, LLREF, and gMUA, and LF versions of RUA, DASA, LLREF, and gMUA, and their associated tradeoffs.

## **1.6 Organization**

The rest of the dissertation is organized as follows. We first overview UA scheduling algorithms for single processors in Chapter 2. In Chapter 3, we discuss lock-based and non-blocking synchronization methods for single-processor UA scheduling, including (space optimal) wait-free and lock-free synchronization

Our real-time scheduling algorithms for multiprocessors, LLREF and gMUA, are presented in Chapters 4 and 5 respectively. Synchronization methods for both algorithms, including lock-based and non-blocking, are discussed in Chapter 6, showing their applicability, performances and trade-offs.

We survey past and related efforts, and contrast them with the dissertation research in Chapter 7. Finally, the dissertation concludes by summarizing its contributions and identifying possible future works in Chapter 8.

## Chapter 2

# Overview of Single Processor UA Scheduling Algorithms

The first public reference of UA scheduling was written by Jensen, Locke, and Tokuda in [47]. The LBESA algorithm presented by Locke in [56] allows almost arbitrary TUF shapes, but it was restricted to resource-independent activities. On the other hand, the DASA presented by Clark allows activities to mutually exclusively share non-CPU resources under the single-unit resource request model<sup>1</sup>. DASA is restricted to downward step TUFs. Both LBESA and DASA yield optimal total utility under downward step TUFs during under-loads.

Recently, several UA scheduling algorithms have been developed (for single processor). Examples include CMA [21], UPA [79], GUS [54], RUA [80], ReUA [83] and CUA. The algorithms consider different TUF shapes, resource models, and scheduling criteria of maximizing the total accrued utility, while allowing most TUF shapes including step and non-step shapes, and concurrent, serialized access to shared objects. Further, the algorithms must yield optimal total utility under downward step TUFs during under-loads. We focus on this scheduling criteria, resource

---

<sup>1</sup>The single-unit resource request model allows a thread to request at most one unit of a resource at any instant. Whereas, the multi-unit resource request model, on the other hand, allows a thread to request more than one unit, at any instant [80].

constraints, and optimality results, as it is of interest to majority of our motivating applications.

Thus, the dissertation considers the RUA and DASA single processor UA scheduling algorithms, as they belong to this category. In this Chapter, we overview RUA and DASA in Section 2.1 and 2.2, respectively.

## 2.1 RUA

RUA considers activities subject to arbitrarily shaped TUF time constraints and concurrent sharing of non-CPU resources including logical (e.g., data objects) and physical (e.g., disks) resources. The algorithm allows concurrent resource sharing under mutual exclusion constraints. RUA's objective is to maximize the total utility accrued by all activities.

RUA's scheduling events include task arrivals, task departures, and lock/unlock requests. When RUA is invoked, it first builds each task's dependency list—that arises due to mutually exclusive object sharing—by following the chain of object requests and ownership. The algorithm then checks for deadlocks by detecting the presence of a cycle in the object/resource graph—a necessary condition for deadlocks. Deadlocks are resolved by aborting that task in the cycle, which will likely contribute the least utility.

After handling deadlocks, RUA examines tasks in the order of non-increasing *potential utility densities* (or PUDs). The PUD of a task is the ratio of the expected task utility to the remaining task execution time, and thus measures the task's return of investment. The algorithm inserts a task and its dependents into a tentative schedule, in the order of their critical times, earliest critical time first. Critical time is the time at which the task TUF has zero utility—i.e., the "deadline" time (all TUFs are assumed to have such a time). The insertion is done by respecting the tasks' dependencies.

After insertion, RUA checks the schedule's feasibility. If infeasible, the inserted task and its dependencies are rejected. The algorithm repeats the process until all tasks are examined and selects

the task at the schedule head for execution.

If a task's critical time is reached and its execution has not been completed, an exception is raised, which is also a scheduling event, and the task is immediately aborted.

## 2.2 DASA

DASA considers activities subject to step-shaped TUFs and concurrent sharing of non-CPU resources under mutual exclusion constraints and under the single-unit resource request model. Thus, DASA's model is a proper subset of RUA: DASA is restricted to step TUFs and the single-unit model while RUA allows arbitrary shaped TUFs and the multi-unit model.

DASA's objective is to maximize the total utility accrued by all activities. DASA's basic operation is identical to that of RUA for the single-unit model. Thus, for the single-unit model, RUA's behavior subsumes DASA's.

## 2.3 Algorithm Differences

In this section, we illustrate the differences of scheduling algorithms including EDF, DASA, and RUA, in terms of the TUF shapes they consider, the feasible utilization demand that they assure, and their scheduling objectives.

The utilization demand is here denoted as  $u/M$ , where  $u$  is the total utilization demand of tasks and  $M$  is the number of processors. (We here consider periodic task sets for convenience, but EDF, DASA, RUA are able to handle non-periodic task sets.) When the utilization demand becomes up to the capacity of processors,  $u/M = 1$ . As we defined previously, we call the case that  $u/M \leq 1$  underload. (We call  $u/M > 1$  overload.) This holds no matter whether the system has single or multiple processors.

TUF time constraint is categorized into step and non-step TUFs. Especially, deadline-based schedul-

ing algorithms, e.g., EDF, does not consider such TUF time constraints. However, since deadline time constraint is a special case of step TUFs as described in Section 1.1, it is illustrated as a subset of step TUFs as in Figure 2.1(a).

(a) EDF

(b) DASA

(c) RUA

**Figure 2.1:** Scheduling Objective Differences Between EDF, DASA, and RUA

Figure 2.1(a) shows EDF's scheduling objective. EDF only considers deadlines and it provides 100% *Deadline Satisfaction Ratio* (or DSR) under underload. It is worth noting that EDF still works with TUF time constraints, although those were not considered in the algorithm design. All figures in this section just show the scheduling objectives that was intended in its design time, but not the algorithm's workability that was not intended.

In Figure 2.1(b), DASA can be viewed as an effort to overcome these restrictions of EDF. Considering step TUF time constraints and overloads, DASA provides 100% DSR optimality under underload and a UA criteria of maximizing total accrued utility under overload. (100% DSR implies 100% AUR in the case where step TUFs are only considered.) On the other hand, RUA is an effort to extend the availability up to non-step TUFs as in Figure 2.1(c).

Chapter 3 discusses synchronization methods for DASA and RUA, including lock-based and non-blocking synchronization.

## Chapter 3

# Synchronization on Single Processors: Lock-Based versus Non-Blocking Methods

The application activities we consider in the dissertation are subject to TUF time constraints, and desire atomic behavior under concurrent, serialized access to shared data objects. Atomic behavior in past UA scheduling algorithms including RUA and DASA is obtained through lock-based synchronization — non-blocking synchronization has not been studied, and therefore we consider it in this dissertation.

Our motivation to consider non-blocking synchronization for UA scheduling is due to our hypothesis that the absence of blocking times, increased concurrency, and reduced system overhead experienced under non-blocking synchronization will result in better optimization of UA criteria than what is possible under lock-based synchronization. The rationale for this hypothesis is that completion time of activities are fundamental to UA scheduling (than they are to deadline scheduling) — they precisely determine activities' attained utility. For example, zero blocking times imply shorter activity sojourn times. This will directly result in greater accrued utility under monotonically decreasing, unimodal TUFs, and greater or the same accrued utility under non-increasing unimodal TUFs.

In this Chapter, we first present our novel wait-free synchronization protocol by focusing on optimizing space costs and without considering any scheduler knowledge<sup>1</sup> in Section 3.1. In Section 3.2, we then develop wait-free versions of RUA and DASA using this protocol. We consider lock-free sharing for UA scheduling in Section 3.3. We also identify the tradeoffs of non-blocking synchronization versus lock-based for UA scheduling 3.3.3.

## 3.1 A Space-Optimal Wait-Free Protocol

### 3.1.1 Algorithm

A wait-free protocol solves the asynchronous single-writer/multiple-reader problem by ensuring that each reader accesses the shared object without any interference from the writer. To realize the wait-free mechanism, the protocol must hold two properties: safety and orderliness [19]. The safety property ensures that the shared object does not become corrupted during reading and writing. The orderliness property ensures that all readers always read the latest data that is completely written by the writer.

The basic idea to achieve the two properties is rooted in the three-slot fully asynchronous mechanism for the single-reader/single-writer problem [20]. For this problem, Chen *et al.* show that three buffers are required to keep the latest completely updated buffer for the next reading, while a writer and a reader are occupying buffers respectively. This mechanism allows that a reader can always obtain data from the buffer slot that is last completely updated, while the writer is writing the new version of the shared data [19].

The buffers needed for the single-writer/multiple-reader problem consist of three types: buffers for readers, a buffer for the latest written data, and a buffer for the next write operation. The buffers for readers must satisfy safety—i.e., sufficient buffers must be available to avoid interference between reading and writing. However, this does not imply that we need as many buffers as there are

---

<sup>1</sup>That is, the presented wait-free scheme is independent of any scheduling algorithm.

readers.

The two buffers for writing are required to realize orderliness—i.e., the latest written data must be saved so that a newly activated reader can access it at any time. In addition, the latest written data must be kept until the writer completely writes the next data into another buffer.

We now discuss how to determine the minimum number of buffers that are needed for the single-writer/multiple-reader problem in the following subsections:<sup>2</sup>

### Protocol Structure and Task Model

Figure 3.1 shows a wait-free protocol’s common implementation. W.2 and R.2 show the code sections of the writer and a reader that write and read data, respectively. W.1 is the code section where the writer decides on the buffer for writing, and updates a control variable that indicates the selected buffer. W.3 is the code section where the writer indicates completion of writing and the buffer that has the latest data. In R.1, the reader checks for the latest data to read.

(a) Write

(b) Read

**Figure 3.1:** Typical Wait-Free Implementation

The buffer size required for the NBW protocol [48] and the improved protocols in [46] is determined based on the temporal properties of tasks. These prior works consider the periodic task model, where tasks concurrently share data objects. Aperiodic tasks are handled by a periodic

---

<sup>2</sup>Note that the space optimality we provide is on the required number of internal buffers, and does not include the control variables needed for the wait-free protocol’s operation. This is because the space cost of internal buffers dominates that of the control variables, especially when the data size becomes larger.

server, so the periodic model is not a limiting assumption. Assuming that all deadlines are met (i.e., during under-load situations and precluding overloads), the maximum number of preemptions of the reader by the writer task in the worst-case can be obtained. We consider the same task model.

In addition, we do not make any assumption on the speed or duration of the read/write operations.

### Number of Buffers in Use

We introduce some notations for convenience, most of which are similar to those in [19]. We denote the total number of readers as  $N^R$  and the  $i^{th}$  reader as  $R_i$ . The writer is simply denoted as  $W$ . The reader  $R_i$ 's  $j^{th}$  instance of reading is denoted as  $R_i^{[j]}$ . The writer's  $k^{th}$  writing instance is denoted as  $W^{[k]}$ .

$R_i^{[j]}(op)$  stands for a specific operation of  $R_i^{[j]}$ . For example,  $R_i^{[j]}(READING[i] = 0)$  implies the execution of one statement in Chen's algorithm [19].  $W^{[k]}(op)$  also stands for the operation in  $W^{[k]}$ . If  $R_i^{[j]}$  reads what  $W^{[k]}$  writes, we denote it as  $w(R_i^{[j]}) = W^{[k]}$ .

As previously mentioned, safety and orderliness can be achieved with multiple buffers for readers, one buffer for the latest written data, and another buffer for the next writing.

**Figure 3.2:** Number of Buffers in Use

Suppose we have 4 readers and 1 writer, as shown in Figure 3.2. In the figure, each horizontal line represents the time interval from start to end of a read or write operation. The operations run concurrently, which includes virtual concurrency on a single processor, or virtual or true concurrency

on multiprocessors. Interference occurs when more than one horizontal line are overlapped at a time point.

At time  $t_1$ ,  $w(R_1)=W^{[2]}$ ,  $w(R_2)=W^{[2]}$ , and  $w(R_3)=W^{[1]}$ . This implies that two buffers are being used by the readers. In addition, one buffer is required to store and save the latest completely written data by  $W^{[4]}$ , and another is needed for the next writing operation by  $W^{[5]}$ . Thus, four buffers are being used in total, at time  $t_1$ .

At time  $t_2$ ,  $w(R_1)=W^{[6]}$ ,  $w(R_2)=W^{[5]}$ ,  $w(R_3)=W^{[4]}$ , and  $w(R_4)=W^{[6]}$ . The latest written data is by  $W^{[6]}$ , and  $W^{[7]}$  is the next operation. Thus, the total number of buffers used at time  $t_2$  is four, which is the minimum number required at  $t_2$  for ensuring safety and orderliness properties.

The basic intuition for determining the minimum number of buffers is to construct a worst-case where the required number of buffers is as large as possible, when the maximum possible number of interferences of all readers with the writer occurs. We map this problem to a problem called the *Diverse Selection Problem* (or DSP) and then solve it.

### Diverse Selection Problem

The DSP denoted as  $D(R, \vec{R}(\vec{x}))$  is defined with the problem range  $R$  and the range vector  $\vec{R}(\vec{x})$  of all elements in the vector  $\vec{x}$ .  $R$  has the lower and upper bounds defined as  $[l, u]$ . Each element  $x_i$  in the vector  $\vec{x}$  has the range  $r_i=[l_i, u_i]$ . The solution to the problem  $D$  is represented as a vector  $\vec{x} = \langle x_1, \dots, x_M \rangle$  where the vector size  $n(\vec{x})$  is  $M$ . Every  $x_i$  must satisfy its range constraint  $r_i$  and the problem range constraint  $R$ . We define  $\{\vec{x}\}$  as a set including all elements of  $\vec{x}$ , but without duplicates. Thus, the size of  $\{\vec{x}\}$ ,  $n(\{\vec{x}\})$ , is less than or equal to  $n(\vec{x})$ . The objective of DSP is to determine the maximum  $n(\{\vec{x}\})$  by selecting  $\vec{x}$ , satisfying all range constraints as diversely as possible.

Given a vector,  $\vec{v} = \langle v_1, \dots, v_i, \dots \rangle$ , we denote the number of  $v_i$ 's having  $k$  value as  $H(\vec{v}, k)$ , and the maximum value among all  $v_i$  elements as  $Top(\vec{v})$ . Given  $D(R, \vec{R}_x)$ , the optimal solution of  $D$  when  $R = [t_1, t_2]$  is denoted as  $n_{[t_1, t_2]}^{max}(\{\vec{x}\})$ .

For example, if  $\vec{v} = \langle 1, 2, 2, 2, 6 \rangle$  then,  $H(\vec{v}, 2) = 3$  and  $Top(\vec{v}) = 6$ . An easy approach to solve DSP is by considering all possible cases. The number of all possible cases is  $n(r_1) \times \dots \times n(r_M)$ , where the  $\vec{R}$  of the problem is given as  $\langle r_1, \dots, r_M \rangle$ . By considering all cases, we can select a vector  $\vec{x}$  that maximizes  $n(\{\vec{x}\})$ . However, such an approach would be computationally expensive.

A more efficient approach to solve DSP can be found by an inductive strategy. Consider a DSP  $D(R, \vec{R}(\vec{x}))$ , where  $R = [1, \infty]$  and  $\vec{R} = \langle [1, u_1], \dots, [1, u_M] \rangle$ . If all lower bounds of elements of  $\vec{x}$  are 1, we can define the upper bound vector  $\vec{u} = \langle u_1, \dots, u_M \rangle$  instead of  $\vec{R}$ , for convenience. In the rest of the section, we call the problem defined by  $D(R, \vec{u})$  as DSP. This is simply because this assumption is well-mapped to the problem of deciding the minimum buffer size for the wait-free protocol.

**Figure 3.3:** An Inductive Approach to DSP

The solution to the problem  $D(R, \vec{u})$  can be represented as  $n_{[1, Top(\vec{u})]}^{max}(\{\vec{x}\})$ . The idea to decompose the problem is shown in Figure 3.3. If the solution to the problem  $D([6, 12], \vec{u})$  can be derived from the solution to the problem  $D([7, 12], \vec{u})$ , we can inductively determine the final solution to the problem  $D([1, 12], \vec{u})$ .

**Theorem 3.1.1** (DSP for the Wait-Free Protocol). *In the DSP  $D(R, \vec{u})$  with  $R = [1, N]$ ,*

$$n_{[t+1]}^{max}(\{\vec{x}\}) = \begin{cases} n_{[t]}^{max}(\{\vec{x}\}) + 1, & \text{if } \sum_{k=0}^{t+1} H(\vec{u}, N - k) > n_{[t]}^{max}(\{\vec{x}\}) \\ n_{[t]}^{max}(\{\vec{x}\}), & \text{otherwise} \end{cases}$$

where  $N = \text{Top}(\vec{u})$ ,  $[t] = [N - t, N]$ , and  $0 \leq t < N$ . When  $t = 0$ , the  $n_{[0]}^{\max}(\{\vec{x}\}) = 1$ .

*Proof.* Assume that we have the solution to the problem  $D([N - t, N], \vec{u})$ . When this problem is extended to  $D([N - (t + 1), N], \vec{u})$ , the ranges of several variables  $x_i$  overlap with the problem range  $[N - (t + 1), N]$ . The number of newly added variables that we need to consider is  $H(\vec{u}, N - (t + 1))$ . When the problem range is extended by 1, the maximum possible increment of  $n_{[t+1]}^{\max}(\{\vec{x}\})$  is 1. The increment happens only if the number of all  $x_i$  which have their  $r_i$  overlapped with  $[N - (t + 1), N]$  is greater than  $n_{[t]}^{\max}(\{\vec{x}\})$ . In other words, this happens when new elements appear in the extended problem scope, or there is an element duplicated within  $[N - t, N]$  at the previous step. Otherwise,  $n_{[t+1]}^{\max}(\{\vec{x}\})$  has no change from before. The increment means that the value of one element is determined as diversely as possible. The proof is by induction on  $t$ .

*Basis.* We show that the theorem holds when  $t = 0$ . When the problem is  $D([\text{Top}(\vec{u}), \text{Top}(\vec{u})], \vec{u})$ , there must be at least one element  $x_i$  with the range  $[1, \text{Top}(\vec{u})]$ , and the maximum possible value of  $n_{[0]}^{\max}(\{\vec{x}\})$  is 1. Hence, the basis for the induction holds.

*Induction step.* Assume that the theorem holds true when  $R = [N - t, N]$ . We arrive at the optimal solution of  $D([N - (t + 1), N], \vec{u})$  with the optimal solution of  $D([N - t, N], \vec{u})$  as in the base step. Suppose that the derived solution  $n_{[t+1]}^{\max}(\{\vec{x}\})$  is not optimal. Then, there must exist another optimal solution  $n_{[t+1]}^{\max}(\{\vec{x}'\})$ . Clearly,  $n_{[t+1]}^{\max}(\{\vec{x}'\})$  is greater than  $n_{[t+1]}^{\max}(\{\vec{x}\})$ . Now, there are two possible cases:

*Case 1.* If  $H(\vec{u}, N - (t + 1)) > n_{[t]}^{\max}(\{\vec{x}\})$ , then  $n_{[t+1]}^{\max}(\{\vec{x}\})$  is  $n_{[t]}^{\max}(\{\vec{x}\}) + 1$ , which is less than  $n_{[t+1]}^{\max}(\{\vec{x}'\})$ . Therefore,  $n_{[t]}^{\max}(\{\vec{x}\}) < n_{[t+1]}^{\max}(\{\vec{x}'\}) - 1$ . This means that there exists another  $\{\vec{x}\}$  that has more than  $n_{[t]}^{\max}(\{\vec{x}\})$  elements. This contradicts the assumption that  $n_{[t]}^{\max}(\{\vec{x}\})$  is optimal.

*Case 2.* If  $H(\vec{u}, N - (t + 1)) = n_{[t]}^{\max}(\{\vec{x}\})$ , then  $n_{[t+1]}^{\max}(\{\vec{x}\})$  is  $n_{[t]}^{\max}(\{\vec{x}\})$ , which is less than  $n_{[t+1]}^{\max}(\{\vec{x}'\})$ . Since no element's range becomes newly overlapped and no element has its duplicate,  $n_{[t+1]}^{\max}(\{\vec{x}'\}) = n_{[t]}^{\max}(\{\vec{x}'\})$ . This means that there exists another  $n_{[t]}^{\max}(\{\vec{x}'\})$ , which is greater than  $n_{[t]}^{\max}(\{\vec{x}\})$ . This contradicts the assumption that  $n_{[t]}^{\max}(\{\vec{x}\})$  is optimal.  $\square$

**Theorem 3.1.2** (Solution Vector for the DSP). *In the DSP  $D(R, \vec{u})$  with  $R = [1, N]$ ,*

$$\{\vec{x}\}_{[t+1]} = \begin{cases} \{\vec{x}\}_{[t]} \cup \{N - (t + 1)\}, & \text{if } \sum_{k=0}^{t+1} H(\vec{u}, N - k) > n_{[t]}^{max}(\{\vec{x}\}), \\ \{\vec{x}\}_{[t]}, & \text{otherwise} \end{cases}$$

where  $N = Top(\vec{u})$ ,  $[t] = [N - t, N]$ , and  $0 \leq t < N$ . When  $t = 0$ ,  $\{\vec{x}\}_{[0]} = \{N\}$ .

*Proof.* By Theorem 3.1.1,  $\{\vec{x}\}$  can be constructed by adding  $\{N - (t + 1)\}$  whenever  $n_t^{max}(\{\vec{x}\})$  increases by 1. Note that this  $\{\vec{x}\}$  is one of the solution vectors.  $\square$

### Similarity to WFBP

The DSP has similarity with the *Wait-Free Buffer size decision Problem* (or WFBP). In this problem, we are given  $M$  readers and their maximum interferences as  $\langle N_1^{max}, \dots, N_M^{max} \rangle$ . The objective of WFBP is to determine the worst-case maximum number of buffers.

**Figure 3.4:** A Worst-Case of the WFBP

Figure 3.4 illustrates how to construct the worst-case where the required number of buffers are as large as possible with an example. For convenience, the index of the writer is reversed compared with Figure 3.2. In this example,  $R_1$ 's maximum interference is 5, which is illustrated in a line. It means that  $w(R_1)$  may belong to the set  $\{W^{[1]}, \dots, W^{[6]}\}$ . We assume that the worst-case happens at time  $t$  between  $W^{[2]}$  and  $W^{[1]}$ , where  $W^{[2]}$  writes the latest completely written data, and  $W^{[1]}$  is the next writing operation for which another buffer is needed.

For this reason, we restate WFBP as determining  $\vec{x} = \langle w(R_1), \dots, w(R_M) \rangle$  that will maximize  $n(\{\vec{x}\} \cup \{W^{[1]}, W^{[2]}\})$ , where  $w(R_i) \in \{W^{[1]}, \dots, W^{[N_i^{max}+1]}\}$ . If we abbreviate  $W^{[j]}$  as  $j$ , the problem is redefined as determining  $\vec{x} = \langle x_1, \dots, x_M \rangle$  that will maximize  $n(\{\vec{x}\} \cup \{1, 2\})$ , where  $x_i \in \{1, \dots, N_i^{max} + 1\}$ .

This is equivalent to DSP except that  $n(\{\vec{x}\} \cup \{1, 2\})$  is used as the objective to maximize, instead of  $n(\{\vec{x}\})$ . Therefore, the final solution  $\{\vec{x}\}$  of a given WFBP is obtained with a sum of the solution from a mapped DSP and a set  $\{1, 2\}$ . We claim that this is correct, because the algorithm for DSP that we propose is designed to find  $\{\vec{x}\}$  which does not have 1 and 2 as its elements, if possible. We can guarantee that in this way, even if the solution from DSP is summed with  $\{1\}$  or  $\{2\}$ , it is still for the worst-case.

**Corollary 3.1.3** (Space Optimality). *If a solution to the WFBP can be obtained, then it must be the minimum and space-optimal buffer size that satisfies the two properties, safety and orderliness.*

*Proof.* The solution is the number of buffers needed in the worst-case of the given problem. Even with one less buffer than the obtained solution, we cannot realize all reading and writing, and still satisfy safety and orderliness. Hence, the solution to the WFBP is the minimum and space-optimal.  $\square$

### Algorithm for WFBP

We now present an algorithm, Algorithm 1, to solve the WFBP based on the previous sections. The algorithm inputs include the number of readers  $M$  and the maximum interference  $N^{max}[i]$ . The *sum* and the function *doesExist(t)* correspond to  $\sum H(\vec{u}, \dots)$  and  $H(\vec{u}, t)$  in Theorem 3.1.1.

To reduce the time complexity of *doesExist(t)*, we sort all  $N^{max}[i]$  before the main loop. *doesExist(t)* uses a static variable, and does not search the entire array  $N^{max}[i]$  each time. The flag  $on_i$  indicates whether or not the DSP solution includes  $i$ . If it does not include 1 or 2, the required buffer size for the WFBP solution,  $n$ , is incremented.

**Algorithm 1:** Algorithm for WFBP

---

```

1 input   : # of readers M; max interference  $N^{max}[M]$ 
2 output : required buffer size  $n$ 
3  $sum=n=0$ ;
4  $on_1=on_2=false$ ;
5 for  $i = 1$  to  $M$  do  $N^{max}[i]++$ ;
6  $sort(N^{max}[1,...,M])$ ;
7 for  $t=N^{max}[1]$  to  $1$  do
8    $sum += doesExist(t, N^{max}[1,...,M])$ ;
9   if  $sum > n$  then
10     $n++$ ;
11    if  $t=2$  then  $on_2 = true$ ;
12    if  $t=1$  then  $on_1 = true$ ;
13 if  $on_2=false$  then  $n++$ ;
14 if  $on_1=false$  then  $n++$ ;

```

---

The time complexity of this algorithm is  $O(M \log M + N^{max})$ . We believe that this cost is reasonable, as the algorithm is run off-line for determining the buffer needs.

**A Wait-Free Implementation**

The NBW protocol uses a circular buffer to realize wait-free synchronization. The idea behind the circular buffer is that while a writer circularly accesses the buffers, the readers follow the writer. However, we cannot use the circular type of buffer because a writer in our protocol needs to determine a safe buffer, which can be any of the buffers.

The same situation arises with Chen's protocol, where the writer can access anywhere. Thus, to implement our protocol, we slightly modify Chen's protocol. Our implementation scheme is shown in Algorithms 2 and 3. In Algorithms 2 and 3, the *GetBuf()* function searches the empty buffer to write to the buffers assigned by Algorithm 1.

Compared with the implementation in [46], our approach does not need to implement separate protocols for "fast" readers and "slow" readers.<sup>3</sup> Additionally, we achieve the speed improvement

---

<sup>3</sup>In [46], readers are classified into fast and slow readers. Fast readers need shorter time for their read operations and thus, the interference during their operations are rare. Slow readers need longer time for their read operations and thus have higher chance of interference during their operations.

---

**Algorithm 2:** Modified Chen's Protocol for Writer

---

```

1  Data: BUFFER [1,...,NB](NB: # of buffers) ; READING [1,...,n] (n: # of readers) ; LATEST
2  GetBuf ()
3  begin
4      bool InUse [1,...,NB];
5      for  $i=1$  to  $NB$  do InUse [i]=false;
6      InUse[LATEST]=true;
7      for  $i=1$  to  $n$  do
8          j = READING [i];
9          if  $j \neq 0$  then InUse [j]=true;
10      $i=1$ ; while InUse [ $i$ ] do ++ $i$ ;
11     return  $i$ ;
12 end

13 Writer ()
14 begin
15     integer widx, i;
16     widx = GetBuf ();
17     Write data into BUFFER [widx];
18     LATEST = widx;
19     for  $i=1$  to  $n$  do
20         Compare-and-Swap(READING [i],0,widx);
21 end

```

---

by reducing the required buffer size, which reduces the number of iterations in *GetBuf()*'s loop, compared with the original Chen's protocol [19].

Note that all algorithms presented in this section are applicable for both single and multi-processor systems, as long as the number of readers and/or the maximum number of interferences that each reader suffers are known. Besides, a task is not restricted to have only a single read or write operation. A simple way to handle the case when a task has multiple read or write operations is to model each read or write operation in a task as a separate reader or writer. Thus, as long as the problem remains a SWMR problem, our algorithms are directly applicable.

**Algorithm 3:** Modified Chen's Protocol for Reader

---

```

1 Data: BUFFER [1,...,NB](NB: # of buffers) ; READING [1,...,n] (n: # of readers) ; LATEST
2 Reader ()
3 begin
4   integer ridx;
5   READING [id]=0;
6   ridx = LATEST;
7   Compare-and-Swap(READING [id],0,ridx);
8   ridx = READING [id];
9   Read data from BUFFER [ridx];
10 end

```

---

**3.1.2 Formal Comparison with Chen's and NBW****Special Case Behavior**

The buffer size that the NBW protocol [48] requires depends on the maximum number of interferences that a reader can suffer from the writer. It does not depend on the number of readers, because simultaneous reading by the readers accesses the same buffer, irrespective of the number of readers.

On the other hand, the buffer size that the Chen's protocol [19] requires is directly proportional to the number of readers, and is independent of the number of interferences. We now show that our protocol subsumes both Chen's protocol and the NBW protocol as special cases.

**Lemma 3.1.4.** *The buffer size for Chen's protocol [19] is a special case of the WFBP solution given in Algorithm 1.*

*Proof.* Assume that we are given  $M$  readers and no information about interferences. We can map this problem to DSP, by setting  $R$  as  $[1, \infty]$  and the upper-bounds of  $\vec{x}$  as  $\langle \infty, \dots, \infty \rangle$ . According to Theorem 3.1.2,  $n(\{\vec{x}\})$  cannot exceed  $n(\vec{x})$ . Thus, the worst-case buffer size is obtained as  $(M + 2)$ , that is  $n(\vec{x}) + n(\{1, 2\})$ . This is exactly the same value as that obtained by Chen's protocol.  $\square$

**Lemma 3.1.5.** *The buffer size for NBW protocol [48] is a special case of the WFBP solution given in Algorithm 1.*

*Proof.* Assume that we are given infinite number of readers with a knowledge of  $Top(\vec{u}) = N^{max}$ . This problem can be modelled as the problem with  $R = [1, N^{max} + 1]$  and  $\forall i, u_i = N^{max} + 1$  for the worst-case. By Theorem 3.1.1,  $H(\vec{u}, N) = \infty$ , and whenever  $t$  increases,  $n(\{\vec{x}\})$  increases by 1 until  $t$  and  $n(\{\vec{x}\})$  reaches to  $N^{max}$  and  $N^{max} + 1$ , respectively. Thus, the worst-case buffer size is obtained as  $N^{max} + 1$ , i.e.,  $n(\{1, \dots, N^{max} + 1\} \cup \{1, 2\})$ . This is exactly the same value as that obtained by NBW protocol.  $\square$

**Theorem 3.1.6** (Upper Bound of the WFBP solution). *In the WFBP,*

$$n^{max}(\{\vec{x}\}) \leq \min(M + 2, N^{max} + 1),$$

where  $M$  is the number of readers and  $N^{max}$  is the maximum number of interferences that a reader can suffer.

*Proof.* Proof follows directly from Lemmas 3.1.4 and 3.1.5.  $\square$

Chen's protocol is attractive because the number of interferences need not be known a-priori. On the other hand, NBW has the advantage that the required number of buffers can be further reduced if the number of interferences are much smaller than the number of readers. Additionally, we note that the number of buffers needed by our algorithm is less than or equal to that of Chen's or NBW protocol.

### Buffer Size Conditions

According to Theorem 3.1.6, our wait-free protocol always finds the number of required buffers which is less than or equal to that of Chen's protocol or the NBW protocol.

We now identify the precise conditions under which the required buffer size of our protocol is equal to that of Chen's or NBW. To derive the conditions, we observe two properties in the WFBP. In the following theorem, we introduce a notation  $\{\{\vec{x}\}\}$ , which denotes the set including all possible solutions  $\{\vec{x}\}$  for the given DSP.

**Theorem 3.1.7** (Chen's Tester). *When the number of readers in the wait-free buffer size decision problem is  $M$  and  $N^{max} > M$ ,*

$$\{3, \dots, M + 2\} \in \{\{\vec{x}\}\}, \text{ if and only if } n^{max}(\{\vec{x}\}) \geq M + 2.$$

*Proof.* We prove both necessary and sufficient conditions.

*Case 1.* Assume that when  $\{3, \dots, M + 2\} \in \{\{\vec{x}\}\}$ ,  $n^{max}(\{\vec{x}\}) < M + 2$ . Since the size of the optimal solution is less than  $M + 2$ , the size of  $\{\vec{x}\}$  cannot exceed  $M + 2$ . This contradicts our assumption that  $\{1, 2\} \cup \{3, \dots, M + 2\}$  is a solution.

*Case 2.* Assume that the set  $\{\vec{x}\}$  is  $\{x_3, \dots, x_{M+2}\}$ , in which  $x_i$ 's are different between each other and aligned in increasing order. Now, all  $x_i$  must not be 1 or 2, otherwise  $n^{max}(\{\vec{x}\})$  is less than  $M + 2$ . Therefore,  $x_3$  should be greater than or equal to 3, and  $x_4$  is greater than  $x_3$ . Inductively,  $x_{i+1} \geq x_i + 1$ , where  $3 \leq i < M + 2$ . In other words, since  $x_i \geq x_{i-1} + 1 \geq x_{i-2} + 2 \geq \dots$ , the inequality  $u_i \geq x_i \geq i$  holds. For this reason,  $\{3, \dots, M + 2\}$  satisfies the range constraints of all elements.  $\square$

By Theorem 3.1.7,  $n^{max}(\{\vec{x}\}) < M + 2$ , if  $\{3, \dots, M + 2\} \notin \{\{\vec{x}\}\}$ . This means that by checking if  $\{3, \dots, M + 2\}$  is feasible for the problem, we can determine whether or not it requires  $M + 2$  buffers that Chen's protocol needs.

**Theorem 3.1.8** (NBW Tester). *When the number of readers in the wait-free buffer size decision problem is  $M$  and  $N^{max} \leq M$ ,*

$$\{2, \dots, N^{max} + 1\} \in \{\{\vec{x}\}\}, \text{ if and only if } n^{max}(\{\vec{x}\}) \geq N^{max} + 1.$$

*Proof.* We prove both necessary and sufficient conditions.

*Case 1.* Assume that when  $\{2, \dots, N^{max} + 1\} \in \{\{\vec{x}\}\}$ ,  $n^{max}(\{\vec{x}\}) < N^{max} + 1$ . Since the size of the optimal solution is less than  $N^{max} + 1$ , the size of  $\{\vec{x}\}$  cannot exceed  $N^{max} + 1$ . This contradicts our assumption that  $\{1, 2\} \cup \{2, \dots, N^{max} + 1\}$  is a solution.

*Case 2.* Assume that the set  $\{\vec{x}\}$  is  $\{x_2, \dots, x_{N^{max}+1}\}$ , in which  $x_i$ 's are different between each other and aligned in increasing order. Now, all  $x_i$  must not be 1, otherwise  $n^{max}(\{\vec{x}\})$  is less than  $N^{max} + 1$ . Therefore,  $x_2$  should be greater than or equal to 2, and  $x_3$  is greater than  $x_2$ . Inductively,  $x_{i+1} \geq x_i + 1$  where  $2 \leq i < N^{max} + 1$ . In other words, since  $x_i \geq x_{i-1} + 1 \geq x_{i-2} + 2 \geq \dots$ , the inequality  $u_i \geq x_i \geq i$  holds. For this reason,  $\{2, \dots, N^{max} + 1\}$  satisfies the range constraints of all elements.  $\square$

We can also investigate if a given WFBP needs  $N^{max} + 1$  buffers or less by checking feasibility with  $\{2, \dots, N^{max} + 1\}$ . We call  $\{3, \dots, M + 2\}$  and  $\{2, \dots, N^{max} + 1\}$  as ‘‘Chen’s tester’’ and ‘‘NBW tester,’’ respectively.

**Figure 3.5:** Decision Procedure

From Theorems 3.1.7 and 3.1.8, we derive a decision procedure that determines the wait-free protocol with the lowest buffer size. Figure 3.5 shows this procedure. To illustrate it, we use the WFBP example in [46], which is also shown in Table 3.1.

By our decision procedure, since  $N^{max} > M$ , Chen’s protocol requires smaller number of buffers than NBW. The next step is determining whether Chen’s tester, which is  $\langle 3, 4, \dots, 9 \rangle$  in this problem, is feasible. It turns out that it is not feasible, as the second element 4 in the tester is out

of the range  $[1, 3]$  of reader 1. Hence, we expect to find smaller number of required buffers than that of Chen's protocol.

**Table 3.1:** Task Set

Task	$N^{max}$
Reader 0	2
Reader 1	2
Reader 2	2
Reader 3	3
Reader 4	3
Reader 5	14
Reader 6	49

Algorithm 1 determines that we need 6 buffers for this problem. We determine a vector  $\{\vec{x}\} = \{1, 2, 3, 4, 15, 50\}$  as a worst-case candidate for the WFBP from Theorem 3.1.2. As mentioned earlier, the solution means that one of the worst-cases occurs when we need buffers for writers  $\{W^{[1]}, W^{[2]}, W^{[3]}, W^{[4]}, W^{[15]}, W^{[50]}\}$ .

### Comparison with Improved Chen's Protocol

In [46], Huang *et al.* suggest a transformation mechanism to reduce the buffer size needs of a given wait-free protocol. The transformation is applied to many wait-free protocols including Chen's protocol. The transformed Chen's protocol is called Improved Chen's protocol in [46].

We cannot formally compare our protocol with Improved Chen's protocol in terms of space cost, because no analytical foundation is given for the transformation mechanism in [46]. In contrast, we present an analytical foundation — theoretical results and algorithms driven by those results — to compute the minimum required space cost of wait-free buffer by considering the safety and orderliness properties needed for the atomicity of wait-free synchronization.<sup>4</sup>

Consequently, a formal comparison with Improved Chen's is not possible, and only an experimental comparison is possible, where the two protocols can be compared for as many cases as possible. We do this in Section 3.1.3. Our experiments in Section 3.1.3 reveal that the buffer size needs of

<sup>4</sup>The work in [46] do not prove that the presented solution is space-optimal, whereas our work does.

our protocol and Improved Chen's are the same, for all the cases that we consider.

Of course, this does not imply that Improved Chen's and ours always need the same number of buffers, because it is impossible that our evaluation studies in Section 3.1.3 cover all the cases. Nevertheless, note that with Corollary 3.1.3, we guarantee that the buffer size needed for wait-free cannot be reduced any further.

Additional advantage of our protocol is that it is not required to divide readers into fast and slow groups and apply two separate reading operations as Improved Chen's does.

### Comparison of Time Complexity

Implementation of NBW and Chen's protocols require the Compare-And-Swap (CAS) instruction. The CAS instruction is used to atomically modify control variables of the wait-free protocol by combining comparison and swap operations into a single instruction. The instruction is available in many modern processors and takes constant time.

NBW has no loop within both write and read operations. However, Chen's protocol has 3 loops within the write operation and no loop within the read operation. With  $n$  buffers, the time complexity of Chen's writing operation is  $O(n)$ .

**Table 3.2:** Asymptotical Time Complexities

<i>Wait-Free Protocol</i>	<i>Read</i>	<i>Write</i>
NBW	$O(1)$	$O(1)$
Chen's	$O(1)$	$O(n)$
Improved Chen's	$O(1)$	$O(n)$
Ours	$O(1)$	$O(n)$

Improved Chen's protocol and our protocol are variations of Chen's protocol, and hence have similar time complexities as that of Chen's writing and reading. According to Theorem 3.1.6, the loop iteration in our protocol's write operation cannot exceed  $M + 2$ . Thus, the time complexity of our protocol is  $O(n)$ , which is the same as that of Chen's. Since the asymptotical speeds are therefore similar, a speed improvement can be obtained (for Chen's, Improved Chen's, and ours) by reducing the buffer size. Table 3.2 summarizes the asymptotical time complexities of the protocols.

### 3.1.3 Numerical Evaluation Studies

We conduct numerical evaluations to evaluate the buffer size needs of our protocol under a broad range of reader/writer conditions, including increasing maximum interferences and readers. We also consider NBW, Chen's, and Improved Chen's protocols for comparative study. We consider Improved Chen's protocol among all protocols in [46], because it is the most space-efficient protocol in [46].

We exclude the Double Buffer protocol [46] from our study as it needs nearly two times the buffer space than Chen's protocol. (The Double Buffer protocol trades off space for time.) Thus, our protocol will clearly outperform the Double Buffer protocol in terms of buffer needs.

#### Increasing Interferences

We consider a task set with 1 writer and multiple readers whose maximum number of interferences  $N_i^{max}$  is randomly generated with a normal distribution (with a fixed standard deviation of 5), and by varying the average. The protocols are evaluated by their buffer size needs — the actual amount of needed memory is the number of buffers times the message size in bytes. Each experiment is repeated 100 times to determine the average buffer sizes.

(a)  $M = 20$

(b)  $M = 30$

(c)  $M = 40$

**Figure 3.6:** Buffer Sizes Under Increasing Interferences With Normal Distribution for  $N_i^{max}$

Figure 3.6 shows the buffer size needs of each protocol as the average  $N_i^{max}$  is increased from

5 to 45, for 20, 30, and 40 readers. From the figure, we observe that as  $N_i^{max}$  increases, the buffer size needs of NBW increases, whereas that of Chen's protocol remains the same (for a given reader size), since its buffer needs is proportional only to the number of readers. As the number of readers increases from 20 to 40, Chen's protocol needs increasing number of buffers. Meanwhile, the number of buffers that our protocol requires never exceeds that of Chen's and NBW's, as Theorem 3.1.6 holds. Interestingly, the number of buffers that Improved Chen's protocol requires is exactly the same as that of ours. Note that no analysis on the buffer size needs of Improved Chen's is presented in [46], whereas Theorem 3.1.6 gives the upper bound on the buffer size needs of our protocol.

(a) Normal Distribution

(b) Uniform Distribution

**Figure 3.7:** Buffer Sizes Under Different  $N_i^{max}$  Distributions with 40 Readers

We observed exact similar trends for other fixed standard deviations for  $N_i^{max}$ 's distribution, and other distributions for  $N_i^{max}$ . Figure 3.7(a) shows the buffer size needs of each protocol, when  $N_i^{max}$  is generated with a normal distribution, with a fixed standard deviation of 10 (instead of 5), and by varying the average  $N_i^{max}$  from 5 to 45, for 40 readers. Figure 3.7(b) shows the protocols' buffer needs under the exact same conditions as those in Figure 3.7(a), except that  $N_i^{max}$  is now generated with an uniform distribution.

From the figures, we observe that our protocol's buffer needs never exceed that of Chen's and NBW's, and is the same as that of Improved Chen's.

### Heterogenous Readers in Multiple Groups

From Figure 3.6, we also observe that when most readers have small  $N_i^{max}$ , the number of buffers needed by our protocol approaches that of NBW's. Moreover, when most readers have larger  $N_i^{max}$ , the number of buffers needed by our protocol approaches that of Chen's protocol's.

This motivates us to study the buffer size needs of our protocol under two groups of readers, one that has small  $N_i^{max}$ 's and the other that has large  $N_i^{max}$ 's. (A similar evaluation is conducted in [46], where readers are classified as "fast" and "slow.") We divide tasks into the two groups whose averages of the (normal) distribution for  $N_i^{max}$ 's are fixed as 5 and 45, respectively. We then vary the ratio of the two groups. For example, 3:1 in the X-axis in Figure 3.8(a) means that the readers having smaller  $N_i^{max}$  are 3 times more than the readers having larger  $N_i^{max}$ .

(a)  $M = 20$ (b)  $M = 30$ (c)  $M = 40$ 

**Figure 3.8:** Buffer Sizes With 2 Reader Groups Under Varying Reader Ratio, for 20, 30, and 40 Readers

Figure 3.8 shows the buffer sizes of each protocol as the ratio is varied from 3:1 to 1:3, for 20, 30, and 40 readers. We observe that the buffers needed for NBW, Improved Chen's, and our protocol increase as the readers with larger  $N_i^{max}$  increases. This result is consistent with that in [46], where Improved Chen's is shown to require less buffers, as fast readers with smaller  $N_i^{max}$  increases. The results confirm that ours and Improved Chen's require the minimum buffer size when considering two heterogenous reader groups.

We now consider a more complex scenario with three reader groups, called "fast," "slow," and "medium," which are not considered in [46]. The averages of the (normal) distribution for  $N_i^{max}$ 's

for the three groups are fixed as 5, 25, and 45, respectively, and the ratio of the three groups are varied from 6:3:1 to 1:3:6. Figure 3.9 shows the results.

(a)  $M = 20$ (b)  $M = 30$ (c)  $M = 40$ 

**Figure 3.9:** Buffer Sizes With 3 Reader Groups Under Varying Reader Ratio, for 20, 30, and 40 Readers

From the figure, we observe that as the number of fast readers increases, the number of buffers needed decreases. Further, we observe that the buffer size required by Improved Chen’s is the same as that of ours even when we include the “medium” reader group in our evaluation.

### 3.1.4 Implementation Experience

A wait-free protocol’s practical effectiveness is determined by its space and time costs. In developing a wait-free protocol, we focus on optimizing space costs, and we establish the space optimality of our protocol. Although reducing the protocol’s time costs is not our goal, we now determine the time costs to establish our protocol’s effectiveness.

Our wait-free protocol (Algorithms 2 and 3) is a modification of Chen’s protocol, augmented with the buffer size computed by Algorithm 1. Thus, we expect that our protocol incurs at most as much time overhead as that of Chen’s. Moreover, the higher space efficiency that our protocol enjoys can lead to higher time efficiency, because it reduces the search space for determining the protocol’s safe buffer—e.g., `GetBuf()`’s loop in Algorithm 2.

To evaluate the actual time costs of our protocol, we implement our protocol in the SHaRK (Soft

Hard Real-Time Kernel) OS [34], running on a 500MHz, Pentium-III processor. Similar to Section 3.1.3, we also implement Chen's, Improved Chen's, and NBW protocols for a comparative study. We also consider lock-based sharing in this study. Note that all protocols in our study can be adopted for both uni-processor and multi-processor systems, although we consider only the performance in the uni-processor in this section.

We consider a task set with 20 readers and a writer, and use a message size of 8 bytes for an inter-process communication (or IPC). We measure the average-case execution time (or ACET) and the worst case execution time (or WCET) for performing an IPC. The execution time for an IPC is the time needed for executing the code segment that accesses the shared object. With traditional lock-based sharing, this code segment is the critical section. Note that a wait-free protocol's IPC execution time includes times for controlling protocol's variables, accessing the shared object, and potential interference from other tasks. Thus, WCET tends to be much larger than ACET.

In Section 3.1.3, we varied the ratio of two reader groups whose averages of the (normal) distribution for  $N_i^{max}$ 's are fixed as 5 and 45, respectively. We now select two cases from which the ratio of readers having smaller and larger  $N_i^{max}$  are 4:1 and 1:4, respectively. These two cases can be represented as 16 fast and 4 slow readers, and 4 fast and 16 slow readers, respectively, for the purpose of Improved Chen's [46], since that protocol needs the readers to be classified as "slow" and "fast". We fix the writer's period as 0.2 msec and let the writer invoke 6,000,000 times during our experiment interval for computing the ACETs. The period of the 20 readers ranges from 400 usec to approximately 10msec.

Figure 3.10 shows the measurements from our implementation. We observe that NBW has the smallest ACET, lock-based sharing has the largest ACET, and Chen's, Improved Chen's, and our protocol have almost the same ACET in our implementation. NBW has the smallest ACET, because its implementation does not have any loop (and thus less computational costs) inside both the reader and writer operations. Lock-based sharing has the largest ACET due to its blocking times. Further, accessing and releasing locks in SHaRK is done through system calls, which takes longer than wait-free protocols (which are implemented without system calls).

(a) 16 Fast and 4 Slow Readers

(b) 4 Fast and 16 Slow Readers

**Figure 3.10:** ACET of Read/Write in SHaRK RTOS

(a) 16 Fast and 4 Slow Readers

(b) 4 Fast and 16 Slow Readers

**Figure 3.11:** WCET of Read/Write in SHaRK RTOS

In [46], when the number of fast readers are increasing, the ACET of Improved Chen's tends to be shorter because the needed buffer size decreases and NBW, a part of Improved Chen's, performs faster. This trend does not appear in our experiments. This is because the expected speed improvement is only (approximately)  $0.1 \text{ } \mu\text{sec}$ . This difference is small enough to be affected by the OS type, code optimizations, and measurement methodology, among other factors. We observed the similar results in WCET in Figure 3.11. Although reducing the protocol's time costs is not our goal, we observe that variations of Chen's including Chen's, Improved Chen's, and ours have much the same ACET and WCET at least in our implementation and thus, we believe that our protocol's time costs is comparable to that of previous protocols.

We have suggested the decision procedure that determines the wait-free protocol having the lowest buffer size in Section 3.1.2. Before applying our protocol, we can determine which protocol, among Chen's, NBW, and ours, requires the least buffer size using the decision procedure described in Figure 3.5. We now apply this decision procedure to the 16 fast/4 slow-reader example considered previously.

Table 3.3 shows 16 fast/4 slow readers'  $N_i^{max}$ 's. At the first step in the decision procedure, we can easily find that  $N^{max} = 47 > M = 20$ . It implies that Chen's protocol needs lower buffer size than NBW. Now, the next step is to check if Chen's tester is feasible. Chen's tester is evaluated as  $\{3, \dots, 22\}$  by Theorem 3.1.7.

**Table 3.3:** Decision procedure on 16 Fast and 4 Slow Readers

Task	$N_i^{max} + 1$	Chen's Tester	Feasibility
Reader 0	48	22	O
Reader 1	47	21	O
Reader 2	47	20	O
Reader 3	47	19	O
Reader 4	10	18	X
Reader 5	9	17	X
Reader 6	9	16	X
Reader 7	9	15	X
Reader 8	8	14	X
Reader 9	7	13	X
Reader 10	7	12	X
Reader 11	6	11	X
Reader 12	6	10	X
Reader 13	4	9	X
Reader 14	3	8	X
Reader 15	3	7	X
Reader 16	3	6	X
Reader 17	3	5	X
Reader 18	3	4	X
Reader 19	3	3	O

Table 3.3 indicates that Chen's tester is not feasible because 18 in the Chen's tester column is not between 1 and 10, for example. Therefore, at the final step, we can conclude that our protocol requires less buffers than Chen's. This is true as shown in Figure 3.12, which shows the number of required buffer size for each protocol.

(a) 16 Fast and 4 Slow Readers

(b) 4 Fast and 16 Slow Readers

**Figure 3.12:** Buffer Sizes

## 3.2 UA Scheduling and Wait-Free Synchronization

The goal of this section is to use the wait-free synchronization presented in Section 3.1, which requires the absolute minimum space costs, for resource sharing under UA scheduling algorithms. Our motivation is that the reduced shared object access time of wait-free synchronization will result in increased activity attained utility. We consider RUA and DASA supporting lock-based sharing mechanisms and their lock-based mechanisms are replaced with our wait-free synchronization for object sharing. We analytically compare RUA's and DASA's wait-free and lock-based versions and establish the fundamental tradeoffs. Our measurements from a POSIX RTOS implementation reveal that during underloads, wait-free algorithms yield optimal utility for step TUFs and significantly higher utility (than lock-based) for non-step TUFs.

### 3.2.1 Synchronization for RUA

#### Lock-Based RUA

RUA's objective is to maximize the total utility accrued by all activities. RUA [80] considers activities subject to arbitrarily shaped TUF time constraints and concurrent sharing of non-CPU resources including logical (e.g., data objects) and physical (e.g., disks) resources. The algorithm

allows concurrent resource sharing under mutual exclusion constraints. (More detail in Chapter 2)

In [80], Wu *et al.* bound the maximum blocking time that a task may experience under RUA for the special-case of the single-unit resource model:

**Theorem 3.2.1** (RUA's Blocking Time). *Under RUA with the single-unit resource model, a task  $T_i$  can be blocked for at most the duration of  $\min(n, m)$  critical sections, where  $n$  is the number of tasks that can block  $T_i$  and have longer critical times than  $T_i$ , and  $m$  is the number of resources that can be used to block  $T_i$ .*

*Proof.* See [80]. □

### Wait-Free RUA

We focus on the single-unit resource model for developing RUA's wait-free version. With wait-free synchronization, RUA is significantly simplified: Scheduling events now include only task arrivals and task departures — lock and unlock requests are not needed anymore. Further, no dependency list arises and need to be built. Moreover, deadlocks will not occur due to the absence of object/resource dependencies. All these reduce the algorithm's complexity. RUA's wait-free version is described in Algorithm 4.

Lock-based RUA's time complexity grows as a function of the number of tasks, while that for wait-free RUA, it grows as a function of the number of readers. (The cost of lock-based RUA is independent of the number of objects, as the dependency list built by RUA may contain all tasks in the worst-case, irrespective of the number of objects.) Because the number of readers cannot exceed the number of tasks, the number of readers can be replaced with the number of tasks.

With  $n$  tasks and  $m$  readers, the time complexity of lock-based RUA is  $O(n^2 \log n)$  [80], while that of wait-free RUA is  $O(n^2 + m)$ , as Algorithm 2, 3, and 4 show. As the number of readers  $m$ , is bounded by  $n$ , wait-free RUA improves upon lock-based RUA from  $O(n^2 \log n)$  to  $O(n^2)$ .

We now formally compare task sojourn times under wait-free and lock-based versions of RUA.

**Algorithm 4:** Wait-Free RUA

---

```

1 input   :  $\mathcal{T}_r$ , task set in the ready queue
2 output : selected thread  $T_{exe}$ 
3 Initialization:  $t=t_{cur}$ ,  $\sigma = \phi$ ;
4 for  $\forall T_i \in \mathcal{T}_r$  do
5   if  $feasible(T_i)$  then
6      $T_i.PUD = \frac{U_i(t+T_i.ExecTime)}{T_i.ExecTime}$ ;
7   else
8      $abort(T_i)$ ;
9  $\sigma_{tmp1} = sortByPUD(\mathcal{T}_r)$ ;
10 for  $\forall T_i \in \sigma_{tmp1}$  from head to tail do
11   if  $T_i.PUD > 0$  then
12      $\sigma_{tmp2} = \sigma$ ;
13      $InsertByDeadline(T_i, \sigma_{tmp2})$ ;
14     if  $feasible(\sigma_{tmp2})$  then
15        $\sigma = \sigma_{tmp2}$ ;
16   else
17     break;
18  $T_{exe} = headOf(\sigma)$ ;
19 return  $T_{exe}$ ;

```

---

We assume that all accesses to lock-based shared objects require  $q$  units of time, and that to wait-free shared objects require  $w$  units. The computation time  $c_i$  of a task  $T_i$  can be written as  $c_i = e_i + m_i \times t_{acc}$ , where  $e_i$  is the computation time excluding accesses to shared objects;  $m_i$  is the number of shared object accesses by  $T_i$ ; and  $t_{acc}$  is the maximum computation time for any object access—i.e.,  $q$  for lock-based objects and  $w$  for wait-free objects.

**Theorem 3.2.2** (Comparison of RUA’s Sojourn Times). *Under RUA, as the critical section  $t_{acc}$  of a task  $T_i$  becomes longer, the difference between the sojourn time with lock-based synchronization,  $s_{lb}$ , and that with wait-free protocol,  $s_{wf}$ , converges to the range:*

$$0 \leq s_{lb} - s_{wf} \leq q \cdot \min(n_i, m_i),$$

where  $n_i$  is the number of tasks that can block  $T_i$  and have longer critical times than  $T_i$ , and  $m_i$  is the number of shared data objects that can be used to block  $T_i$ .

*Proof.* The sojourn time of a task  $T_i$  under RUA with lock-based synchronization includes the execution time  $c_i = e_i + m_i \times q$ , the preemption time  $I_i$ , and the blocking time  $B_i$ . Based on Theo-

rem 3.2.1, the blocking time  $B_i$  is at most  $q \times \min(n_i, m_i)$ . On the other hand, the sojourn time of task  $T_i$  under RUA with wait-free protocol does not include any blocking time. Therefore, the difference between lock-based synchronization and wait-free is at most  $m_i \times (q-w) + q \times \min(m_i, n_i)$ . Assuming that the data for synchronization is large enough such that the execution time difference in the algorithm between lock-based synchronization and wait-free becomes negligible (i.e., the time for reading and writing the data object becomes dominant in the time for synchronization), then  $q$  and  $w$  converge and become equal. In this case, the sojourn time of  $T_i$  under lock-based synchronization is longer than that under wait-free by  $q \times \min(n_i, m_i)$ .  $\square$

The reduced sojourn time of a task under wait-free increases the accrued utility of the task, for non-increasing TUFs. Further, this potentially allows greater number of tasks to be scheduled and completed before their critical times, yielding more total accrued utility (for such TUFs).

Based on Theorem 3.2.2, we can now estimate the difference in the Accrual Utility Ratio (or AUR) between wait-free and lock-based, for non-increasing TUFs. AUR is the ratio of the actual accrued utility to the maximum possible utility.

**Corollary 3.2.3.** *Under RUA, with non-increasing TUFs, as the critical section  $t_{acc}$  of a task  $T_i$  becomes longer, the difference in AUR,  $\Delta AUR = AUR_{wf} - AUR_{lb}$ , between lock-based synchronization and wait-free converges to:*

$$0 \leq \Delta AUR \leq \sum_{i=1}^N \frac{U_i(s_{wf}) - U_i(s_{wf} + q \cdot \min(m_i, n_i))}{U_i(0)},$$

where  $U_i(t)$  denotes the utility accrued by task  $T_i$  when it completes at time  $t$ , and  $N$  is the number of tasks.

*Proof.* It directly follows from Theorem 3.2.2.  $\square$

When the data for synchronization is small and the time for reading and writing the shared data object is not dominant,  $q$  and  $w$  are more dependent on the object sharing algorithm. The execution time of lock-based synchronization,  $q$ , includes the time for the locking and unlocking procedure

(needed for mutual exclusion), executing the scheduler, and accessing the shared object, while  $w$  includes the time for controlling the wait-free protocol's variables and accessing the shared object. The wait-free protocol does not activate the scheduler, and hence avoids significant system overhead. Consequently,  $w$ 's of many wait-free protocols are known to be shorter than  $q$  [23, 46]. Thus, no matter what the size of the data to communicate between tasks is, wait-free synchronization conclusively accrues more utility compared with lock-based synchronization.

### 3.2.2 Synchronization for DASA

Like RUA, DASA's objective is to maximize the total utility accrued by all activities. DASA's basic operation is identical to that of RUA for the single-unit model (see Section 3.2.1). Thus, for the single-unit model, RUA's behavior subsumes DASA's. Therefore, in [80], Wu *et al.* also show that the maximum blocking time that a task may suffer under DASA is identical to that under RUA (for the single-unit model), which is stated in Theorem 3.2.1.

#### Wait-Free DASA

Since TUFs under DASA are step-shaped, shorter sojourn time does not increase accrued utility, as tasks accrue the same utility irrespective of when they complete before their critical times, as long as they do so. However, shorter sojourn time is still beneficial, as it reduces the likelihood of missing task critical times. Hence, lesser the number of tasks missing their critical times, higher will be the total accrued utility.

Wait-free synchronization prevents DASA from losing utility no matter how many dependencies arise. Similar to wait-free RUA, wait-free DASA is also simplified: Scheduling events now include only arrivals and departures, no dependency list needs to be built, and no deadlocks will occur.

The time complexities of lock-based and wait-free DASA are identical to that of lock-based and wait-free RUA:  $O(n^2)$  [27] and  $O(n^2+m)$ , respectively. Similar to wait-free RUA, as  $m$  is bounded by  $n$ , wait-free DASA improves upon lock-based DASA from  $O(n^2 \log n)$  to  $O(n^2)$ .

The comparison of DASA's sojourn times under lock-based and that under wait-free is similar to Theorem 3.2.2:

**Theorem 3.2.4** (Comparison of DASA's Sojourn Times). *Under DASA, as the critical section  $t_{acc}$  of a task  $T_i$  becomes longer, the difference between the sojourn time with lock-based synchronization,  $s_{lb}$ , and that with wait-free protocol,  $s_{wf}$ , converges to:*

$$0 \leq s_{lb} - s_{wf} \leq q \times \min(n_i, m_i),$$

where  $n_i$  is the number of tasks that can block  $T_i$  and have longer critical times than  $T_i$ , and  $m_i$  is the number of shared data objects that can be used to block  $T_i$ .

*Proof.* See proof of Theorem 3.2.2. □

### 3.2.3 Implementation Experience

We implemented the lock-based and wait-free algorithms in the *meta-scheduler* scheduling framework [52], which allows middleware-level real-time scheduling atop POSIX RTOSes. We used QNX Neutrino 6.3 RTOS running on a 500MHz, Pentium-III processor in our implementation.

Our task model for computing the wait-free buffer size follows the model in [46]. We determine the maximum number of times the writer can interfere with the reader,  $N_R^{max}$  as:

$$N_R^{max} = \max \left( 2, \left\lceil \frac{p_R - (c - c_R)}{p_W} \right\rceil \right).$$

$p_R$  and  $p_W$  denote the reader's and the writer's period, respectively. For simplicity, we set the task critical times to be the same as the task periods.  $c$  is the reader's worst-case execution time, and  $c_R$  is the time needed to perform a read operation.

Table 3.4 shows our task parameters. We consider a task set of 10 tasks, which includes 5 readers and 5 writers, mutually exclusively sharing at most 5 data objects. We set  $c_R$  for each object to be approximately 20% of  $c$  for each reader, which implies that tasks share large amounts of data. We allow a reader to read from 1 object to at most 5 objects. We also vary the number of readers from

**Table 3.4:** Experimental Parameters for Tasks

<i>Name</i>	<i>p(msec)</i>	<i>c(msec)</i>	<i>Shared Objects</i>
Writer1	100	10	$r_1$
Writer2	100	10	$r_2$
Writer3	100	10	$r_3$
Writer4	100	10	$r_4$
Writer5	100	10	$r_5$
Reader1	900	100	$r_1 r_2 r_3 r_4 r_5$
Reader2	1000	100	$r_2 r_3 r_4 r_5 r_1$
Reader3	1100	100	$r_3 r_4 r_5 r_1 r_2$
Reader4	1200	100	$r_4 r_5 r_1 r_2 r_3$
Reader5	1300	100	$r_5 r_1 r_2 r_3 r_4$

1 to 5. The minimum buffer size needed for the wait-free protocol is calculated by Algorithm 1. The actual amount of memory needed is the buffer size multiplied by the message size in bytes.

For each experiment, we generate approximately 16,000 tasks and measure the AUR and the CMR (or critical time meet ratio) under RUA's and DASA's lock-based and wait-free versions, where CMR is the ratio of the number of tasks that meet their critical times to the total number of task releases. The performance comparison between both are shown with the varying number of shared objects and the varying number of tasks.

(a) Lock-Based

(b) Wait-Free

**Figure 3.13:** Performance of Lock-Based and Wait-Free DASA Under Increasing Number of Shared Objects

Figure 3.13 and Figure 3.14 show lock-based and wait-free DASA's AUR and CMR (on the right-side y-axes), under increasing number of shared data objects and under increasing number of reader

tasks, respectively. The figures also show (on the left-side y-axes) the number of buffers used by lock-based and wait-free, as well as the number of task blockings' and task abortions that occur under lock-based.

From Figures 3.13(a) and 3.14(a), we observe that the AUR and CMR of lock-based DASA decrease as the number of shared objects and number of readers increase. This is because, as the number of shared objects and readers increase, greater number of task blockings' occurs, resulting in increased sojourn times, critical time-misses, and consequent abortions.

(a) Lock-Based

(b) Wait-Free

**Figure 3.14:** Performance of Lock-Based and Wait-Free DASA Under Increasing Number of Readers

Wait-free DASA is not subject to this behavior, as Figures 3.13(b) and 3.14(b) show. Wait-free DASA achieves 100% AUR and CMR even as the number of shared objects and readers increase. This is because, wait-free eliminates task blockings'. Consequently, the algorithm produces a critical time (or deadline)-ordered schedule, which is optimal for under-load situations (i.e., all critical times are satisfied). Thus, all tasks complete before their critical times, and the algorithm achieves the maximum possible total utility.

However, the number of buffers needed for wait-free (calculated by Algorithm 1) increases as the number of objects and readers increase. But Algorithm 1 is space-optimal — the needed buffer size is the absolute minimum possible.

Similar to the DASA experiments, we compare the performance of RUA's lock-based and wait-free

(a) Lock-Based

(b) Wait-Free

**Figure 3.15:** Performance of Lock-Based and Wait-Free RUA Under Increasing Number of Shared Objects

versions. Since RUA allows arbitrarily-shaped TUFs, we consider a heterogeneous class of TUF shapes including step, parabolic, and linearly-decreasing. The results are shown in Figures 3.15 and 3.16. We observe similar trends as that of DASA's: Lock-based RUA's AUR and CMR decrease as the objects and readers increase, due to increased task blockings'.

(a) Lock-Based

(b) Wait-Free

**Figure 3.16:** Performance of Lock-Based and Wait-Free RUA Under Increasing Number of Readers

Similar to DASA, wait-free sharing alleviates task blockings' under RUA. Consequently, RUA produces optimal-schedules during under-load situations in terms of meeting all task critical times, and thus yields 100% CMR.

RUA with wait-free sharing yields greater AUR than that under lock-based, as objects and readers increase. However, unlike DASA, RUA does not yield 100% AUR with wait-free, because tasks accrue different utility depending upon when they complete, even when they do so before task critical times (due to non-step shapes). Further, the algorithm does not necessarily complete tasks at their optimal times as that scheduling problem is  $\mathcal{NP}$ -hard (and RUA is a heuristic).

### 3.3 UA Scheduling and Lock-Free Synchronization

In this section, we focus on lock-free synchronization of *dynamic* embedded real-time systems on a single processor. By dynamic systems, we mean those subject to resource overloads due to context dependent activity execution times and arbitrary activity arrivals. To account for the variability in activity arrivals, we describe arrival behaviors using the *unimodal arbitrary arrival model* (UAM) [40]. UAM specifies the maximum number of activity arrivals that can occur during any time interval. Consequently, the model subsumes most traditional arrival models (e.g., periodic, sporadic) as special cases.

We show in the previous section that wait-free sharing is suitable to the real-time system including UA scheduling. By definitions, wait-free is much stronger than lock-free in the sense that it does not allow any operation on the wait-free object to starve forever. However, it is not clear whether the wait-free object can be applied to the dynamic environment such as UAM.

We consider lock-free sharing under the UAM. The past work on lock-free sharing upper bounds the retries under restrictive arrival models like periodic [6]—retry bounds under the UAM are not known. Moreover, we consider the UA criteria of maximizing the total utility, while allowing most TUF shapes including step and non-step shapes, and mutually exclusive concurrent object sharing. We focus on the RUA and DASA for that model.

We derive the upper bound on lock-free RUA's and DASA's retries under the UAM. Since lock-free sharing incurs additional time overhead due to the retries (as compared to lock-based), we establish the conditions under which activity sojourn times are shorter under lock-free RUA and DASA than

under lock-based, for the UAM. From this result, we establish the maximum increase in activity utility under lock-free over lock-based. Further, we implement lock-free and lock-based RUA on a POSIX RTOS.

### 3.3.1 Wait-Free Buffer under UAM

Under the UAM where activities arbitrarily arrive and concurrently share data objects, the multi-writer multi-reader (or MWMR) problem always occurs, even though a single activity(or task) is a writer. This is because, under the UAM, multiple jobs of a single writer task can be simultaneously pending in the ready queue and be requesting the same shared object. This situation will not occur under a periodic task model during under-loads if considering the single-writer multi-reader (or SWMR) problem. For this reason, the MWMR wait-free buffer should be considered.

Secondly, for the UAM, the wait-free buffer should be anonymous in the sense that the read/write operations should not require knowledge of the identity of the jobs. This is because the UAM allows multiple jobs of the same task to inherit the same task identity and to access a shared object. In this case, two jobs having same task identify may access the same memory place at the same time, which results in conflict. Many number of wait-free buffer mechanisms, however, are typically established under the assumption that read/write operations know the identity of tasks invoking the operations [18, 38, 77]. These non-anonymous mechanisms does not handle the aforementioned case under UAM.

Further, wait-free buffer for the MWMR problem should overcome inherent space-inefficiency. Theoretically, with  $n$  readers and  $m$  writers, the minimum number of needed buffers is  $n + m + 1$  for the MWMR problem [66]. However, most wait-free solutions for the MWMR problem need  $n \times m$  number of atomic buffers, where each atomic buffer solves the single-writer/single-reader (SWSR) problem [38, 77]. The atomicity of the buffer is ensured by using multiple buffers, which further increases the number of needed buffers. To make things worse, the UAM allows greater number of concurrent operations than more restrictive arrival models like the periodic considered

in past works [46, 48], resulting in even greater number of buffers.

These requirements are not only for wait-free buffer, but also for all non-blocking sharing including lock-free sharing. Several lock-free objects exist for the MWMM problem [58], which are anonymous and space-efficient. However, few wait-free buffers exist, which support all requirements above. The wait-free buffer presented in [5] was based on specific system models, such as quantum-based and priority-based, although it requires the optimal buffer size for MWMM problem. Another wait-free buffer proposed in [70] is limited to the system where writers have the highest priority on the host processor and no two writers execute on one host processor.

### 3.3.2 Bounding Retries Under UAM

Figure 3.17 shows the three dimensions of the task model that we consider. The first dimension is the arrival model. We consider the UAM, which is more relaxed than the periodic model, but more regular than the aperiodic model. Hence it falls in between these two ends of the (regularity) spectrum of the arrival dimension. For a task  $T_i$ , its arrival using UAM is defined as a tuple  $\langle l_i, h_i, W_i \rangle$ , meaning that the maximal number of job arrivals of the task during any sliding time window  $W_i$  is  $h_i$  and the minimal number is  $l_i$  [40]. Jobs may arrive simultaneously. The periodic model is a special case of UAM with  $\langle 1, 1, W_i \rangle$ .

**Figure 3.17:** Three Dimensions of Task Model

We refer to the  $j^{\text{th}}$  job (or invocation) of task  $T_i$  as  $J_{i,j}$ . The basic scheduling entity that we consider

is the job abstraction. Thus, we use  $J$  to denote a job without being task specific, as seen by the scheduler at any scheduling event.

A job's time constraint, which forms the second dimension, is specified using a TUF. A TUF subsumes deadline as a special case (i.e., binary-valued, downward step TUF). Jobs of a task have the same TUF.  $U_i(\cdot)$  denotes task  $T_i$ 's TUF; thus, completion of  $T_i$  at time  $t$  will yield  $U_i(t)$  utility.

TUFs can take arbitrary shapes, but must have a (single) "critical time." Critical time is the time at which the TUF has zero utility. For the special case of deadlines, critical time corresponds to the deadline time. In general, the TUF has zero utility after the critical time. We denote the critical time of task  $i$ 's  $U_i(\cdot)$  as  $D_i$ , and assume that  $D_i \leq W_i$ .

The third dimension is the timeliness optimization objective. Example objectives include satisfying all critical times for step TUFs during under-loads, and maximizing total accrued utility for arbitrarily-shaped TUFs during under-loads and overloads. Our model includes the UAM, TUFs, and the objective of maximizing total utility.

Finally, the resource model we consider here does not allow nested critical sections, which may cause deadlock in lock-based synchronization, and on the other hand, complicates implementation in lock-free synchronization.

### Preemptions Under UA Schedulers

Under fixed priority schedulers such as rate-monotonic (RM), a lower priority job can be preempted at most once by each higher priority job if no resource dependency (that arises due to concurrent object sharing) exists. This is because a job does not change its execution eligibility until its completion time. However, under UA schedulers such as RUA, execution eligibility of a job dynamically changes.

In Figure 3.18, assume that two jobs  $J_{1,1}$  and  $J_{2,1}$  arrive at time  $t_0$ , and scheduling events occur at  $t_1$ ,  $t_2$ , and  $t_3$ .  $J_{2,1}$  occupies CPU at  $t_0$  and is preempted by  $J_{1,1}$  at  $t_1$ . Subsequently,  $J_{1,1}$  is preempted by  $J_{2,1}$  at  $t_2$ , which does not happen under RM scheduling. Under RUA, a simple

**Figure 3.18:** Mutual Preemption Under RUA

condition causing this *mutual preemption*, where a job which has preempted another job can be subsequently preempted by the other job, is when TUFs of the two jobs are increasing.

This potential mutual preemption distinguishes RUA from traditional schedulers such as RM, where a job can be preempted by another job at most once. Hence, the maximum number of preemptions under RM that a job may suffer can be bounded by the number of releases of other jobs during a given time interval (see [5]). However, this is not true with RUA, as one job can be preempted by another job more than once. Thus, the maximum number of preemptions that a job may experience under RUA is bounded by the number of events that invoke the scheduler.

**Lemma 3.3.1** (Preemptions Under UA scheduler). *During a given time interval, a job scheduled by a UA scheduler can experience preemptions by other jobs at most the number of the scheduling events that invoke the scheduler.*

*Proof.* Assume that preemptions happen more than the scheduling events during a given time interval. It means that preemptions can occur without invoking the scheduler, which is not true.  $\square$

Lemma 3.3.1 helps compute the upper bound on the number of retries on lock-free objects for our model. This is also true with other UA schedulers [64], because it is impossible for more preemptions to occur than scheduling events. Especially, when lock-free synchronization and RUA are considered, the scheduling events only include job arrivals and departures, but not lock and unlock requests.

### Bounded Retry Under UAM

Under our model, jobs with TUF constraints arrive under the UAM, and there may not always be enough CPU time available to complete all jobs before their critical times. We now bound lock-free RUA's retries under this model.

**Theorem 3.3.2** (Lock-Free Retry Bound Under UAM). *Let jobs arrive under the UAM  $< 1, h_i, W_i >$  and are scheduled by RUA (or DASA). When a job  $J_i$  accesses more than one lock-free object,  $J_i$ 's total number of retries,  $f_i$ , is bounded as:*

$$f_i \leq 3h_i + \sum_{j=1, j \neq i}^N 2h_j \left( \left\lceil \frac{D_i}{W_j} \right\rceil + 1 \right)$$

where  $N$  is the number of tasks.

*Proof.* In Figure 3.19,  $J_i$  is released at time  $t_0$  and has the absolute critical time  $(t_0 + D_i)$ . After the critical time,  $J_i$  will not exist in the ready queue, because it will be either completed by that time or aborted by RUA. Thus, by Lemma 3.3.1, the number of retries of  $J_i$  is bounded by the maximum number of all scheduling events that occur within the time interval  $[t_0, t_0 + D_i]$ . The scheduling events that  $J_i$  suffers can be categorized into those occurring from other tasks,  $T_j, j \neq i$  and those occurring from  $T_i$ . We consider these two cases:

**Figure 3.19:** Interferences under UAM

*Case 1* (Scheduling events from other tasks): To account for the worst-case event occurrence where the maximum number of scheduling events of  $T_j$  occurs within  $W_i$ , we assume that all instances of  $T_j$  in the window  $W_j^1$  are released right after time  $t_0$ , and all instances of  $T_j$  in the window  $W_j^3$  are

released before time  $(t_0 + D_i)$ . Thus, the maximum number of releases of  $T_j$  within  $[t_0, t_0 + D_i]$  is  $\lceil D_i/W_j \rceil + 1$ . It also holds when  $W_j > D_i$  as  $\lceil D_i/W_j \rceil + 1 = 2$ .

All jobs of  $T_j$  before the first window  $W_j$  must depart either by completion or by abortion before  $t_0$ . All released jobs must be completed or aborted, so that the number of scheduling events that a job can create is at most 2. Hence,  $h_j (\lceil D_i/W_j \rceil + 1)$  is multiplied by 2.

*Case 2* (Scheduling events from the same task): In the worst-case where the maximal scheduling events by  $T_i$  occurs within  $W_i$ , all jobs of  $T_i$  are released and completed within  $[t_0, t_0 + D_i]$ , which results in at most  $2h_i$  events.  $T_i$ 's jobs, which are released during  $[t_0 - D_i, t_0]$  also cause events within  $[t_0, t_0 + D_i]$  by completion. Thus, the total number of events that are possible is  $3h_i$ .

The sum of case 1 and case 2 is the upper bound on the number of events. It is also the maximum number of total retries of  $J_i$ 's objects.  $\square$

Note that  $f$  has nothing to do with the number of lock-free objects in  $J_i$  in Theorem 3.3.2, even when  $J_i$  accesses more than one lock-free object. This is because no matter how many objects  $J_i$  accesses,  $f$  cannot exceed the number of events. Further, even if  $J_i$  accesses a single object, the retry can occur only as many times as the number of events.

Theorem 3.3.2 also implies that the sojourn time of  $J_i$  is bounded. The sojourn time of  $J_i$  is computed as the sum of  $J_i$ 's execution time, the interference time<sup>5</sup> by other jobs, the lock-free object accessing time, and  $f$  retry time.

### 3.3.3 Lock-Based versus Lock-Free

We now formally compare lock-based and lock-free sharing by comparing job sojourn times. We do so, as sojourn times directly determine critical time-misses and accrued utility. The compari-

---

<sup>5</sup>Interference time of a task is the time that the task has to wait to get the processor for execution (since its arrival) as the processor is busy executing other tasks (since those tasks were deemed by the scheduler as more eligible to execute). Thus, for a task  $T_i$ , its interference time is the time spent on executing other tasks when task  $T_i$  is runnable.

son will establish the tradeoffs between lock-based and lock-free sharing: Lock-free is free from blocking times on shared object accesses and scheduler activations for resolving those blockings, while lock-based suffers from these. However, lock-free suffers from retries, while lock-based does not.

We assume that all accesses to lock-based objects require  $q$  time units, and to lock-free objects require  $s$  time units. The computation time  $c_i$  of a job  $J_i$  can be written as  $c_i = e_i + m_i \times t_{acc}$ , where  $e_i$  is the computation time not involving accesses to shared objects;  $m_i$  is the number of shared object accesses by  $J_i$ ; and  $t_{acc}$  is the maximum computation time for any object access—i.e.,  $q$  for lock-based objects and  $s$  for lock-free objects.

The worst-case sojourn time of a job with lock-based is  $e_i + I + q \cdot m_i + B$ , where  $B$  is the worst-case blocking time and  $I$  is the worst-case interference time. In [80], it is shown that a job  $J_i$  under RUA can be blocked for at most  $\min(m_i, n_i)$  times, where  $n_i$  is the number of jobs that could block  $J_i$  and  $m_i$  is the number of objects that can be used to block  $J_i$ . Thus,  $B$  can be computed as  $q \cdot \min(m_i, n_i)$ . On the other hand, the worst-case sojourn time of a job with lock-free is  $e_i + I + s \cdot m_i + T^R$ , where  $T^R$  is the worst-case retry time.  $T^R$  can be computed as  $s \cdot f$  by Theorem 3.3.2. Thus, the difference between  $q \cdot m_i + B$  and  $s \cdot m_i + T^R$  is the sojourn time difference between lock-based and lock-free.

Based on the notation, we formally compare lock-based and lock-free sharing under our model.

**Theorem 3.3.3** (Lock-Based versus Lock-Free Sojourn Time). *Let jobs arrive under the UAM and be scheduled by RUA (or DASA). If*

$$\begin{cases} \frac{s}{q} < \frac{2}{3}, & \text{when } m_i \leq n_i \\ \frac{s}{q} < \frac{m_i + n_i}{m_i + 3h_i + 2x_i}, & \text{when } m_i > n_i, \end{cases}$$

then  $J_i$ 's maximum sojourn time with lock-free is shorter than that with lock-based, where  $x_i = \sum_{j=1, j \neq i}^N h_j \left( \left\lceil \frac{D_j}{W_j} \right\rceil + 1 \right)$ .

*Proof.* Let  $X$  denote  $q \cdot m_i + q \cdot \min(m_i, n_i)$  and  $Y$  denote  $s \cdot m_i + s \cdot f_i$  to compare sojourn times.  $n_i$  is bounded by the total number of jobs that may happen during executing  $J_i$  so that  $n_i$  is at most

$2h_i + \sum_{j=1, j \neq i}^N h_j (\lceil D_i/W_j \rceil + 1) = 2h_i + x_i$ . We now search for the condition when lock-free sharing yields shorter sojourn times than lock-based, which means  $X > Y$ . There are two cases:

*Case 1:* When  $m_i \leq n_i$ ,  $X$  becomes  $2q \cdot m_i$ .  $Y$  is  $s(m_i + 3h_i + 2x_i)$ . Therefore:

$$Y = \frac{s}{2q}X + s(3h_i + 2x_i),$$

where  $X = 2qm_i \leq 2qn_i \leq 2q(2h_i + x_i)$ . The condition that leads to  $X > Y$  can be derived only when  $\frac{s}{2q} < 1$ . Otherwise,  $Y$  is always greater than  $X$ , which means that the sojourn time with lock-free objects is greater than that with lock-based objects. Assuming  $\frac{s}{2q} < 1$ ,  $X$  and  $Y$  become the same when  $X = \frac{2qs(3h_i+2x_i)}{2q-s}$ . Hence, when  $\frac{2qs(3h_i+2x_i)}{2q-s} < X$ ,  $X$  is greater than  $Y$ , where  $X \leq 2q(2h_i + x_i)$ . This leads to:

$$\frac{q}{s} > \frac{1}{2} + \frac{3h_i + 2x_i}{2m_i}.$$

Since  $m_i \leq 2h_i + x_i$ ,

$$\frac{q}{s} > \frac{1}{2} + \frac{3h_i + 2x_i}{2m_i} \geq \frac{1}{2} + \frac{3h_i + 2x_i}{4h_i + 2x_i} \geq \frac{3}{2} - \frac{1}{4 + 2\frac{x_i}{h_i}}.$$

Therefore, the sufficient condition is  $\frac{q}{s} > \frac{3}{2}$ .

*Case 2:* When  $m_i > n_i$ ,  $X$  becomes  $q(m_i + n_i)$  and we can obtain  $Y = \frac{s}{q}X + s(3h_i + 2x_i - n_i)$ . To make  $X > Y$ ,  $\frac{s}{q}$  should be less than 1. Otherwise, the sojourn time of jobs with lock-based objects is always less than that with lock-free objects.  $X$  converges to  $Y$  at  $X = \frac{qs(3h_i+2x_i-n)}{q-s}$ .

Therefore,

$$\frac{qs(3h_i + 2x_i - n_i)}{q - s} < q(m_i + n_i).$$

This inequality leads to:

$$\frac{s}{q} < \frac{m_i + n_i}{m_i + 3h_i + 2x_i}.$$

This implies that  $\frac{s}{q} < 1$ , since  $\frac{m_i+n_i}{m_i+3h_i+2x_i} < 1$  considering  $n_i \leq 2h_i + x_i$ . □

Theorem 3.3.3 states that no matter whether  $m_i$  is greater than  $n_i$ ,  $\frac{s}{q} < 1$  is necessary for jobs to obtain a shorter sojourn time under lock-free (since  $\frac{m_i+n_i}{m_i+3h_i+2x_i}$  is less than one when  $m_i > n_i$ ).

Especially when  $m_i \leq n_i$ ,  $\frac{s}{q} < 2/3$  is sufficient for jobs to obtain a shorter sojourn time under lock-free. Shorter sojourn times always yield higher utility with non-increasing TUFs, but not always with increasing TUFs. However, it is likely to improve performance at the system-level even with increasing TUFs, because each job can save more CPU time for other jobs.

In [5], Anderson *et al.* show that  $s$  is often much smaller than  $q$  in comparison with the vast majority of lock-free objects in most systems, such as buffers, queues, and stacks. It also is known that lock-free approaches work particularly well for such simple objects. Much smaller access time of lock-free approaches is in line with the results of our experiments in Section 3.3.4, where we consider UAM arrival and RUA scheduling. Thus, lock-free sharing is very attractive for UA scheduling as most UA schedulers' object sharing mechanisms have higher time complexity.

Theorem 3.3.3 does not reveal anything regarding aggregate system-level performance, but only job-level performance. Since RUA's objective is to maximize total utility, we now compare lock-based and lock-free sharing in terms of *accrued utility ratio* (or AUR). AUR is the ratio of the actual accrued total utility to the maximum possible total utility. The AURs of lock-based and lock-free sharing under RUA are bounded as follows:

**Lemma 3.3.4** (AUR of Lock-free RUA). *Let the  $i^{\text{th}}$  task's jobs arrive under the UAM,  $\langle l_i, h_i, W_i \rangle$ , and be scheduled by RUA. If all jobs are feasible and their TUFs are non-increasing, then the AUR of lock-free sharing, AUR, over time converges into:*

$$\frac{\sum_{i=1}^N \frac{l_i}{W_i} U_i(e_i + s \cdot m_i + I_i + R_i)}{\sum_{i=1}^N \frac{l_i}{W_i} U_i(0)} < AUR < \frac{\sum_{i=1}^N \frac{h_i}{W_i} U_i(e_i + s \cdot m_i)}{\sum_{i=1}^N \frac{h_i}{W_i} U_i(0)},$$

where  $U_i(t)$  is task  $i$ 's utility at time  $t$ , and  $N$  is the number of tasks.

*Proof.* The AUR is computed as  $\sum_{i=1}^N n_i \cdot U_i(s_i^{lf}) / \sum_{i=1}^N n_i \cdot U_i(0)$ , where  $s_i^{lf}$  is the sojourn time of  $T_i$  and  $n_i$  is the number of jobs of  $T_i$  completed within a time interval. Under UAM,  $n_i$  is within the range of  $[l_i \lfloor \frac{\Delta t}{W_i} \rfloor, h_i (\lceil \frac{\Delta t}{W_i} \rceil + 1)]$  in a time interval  $\Delta t$ , i.e.,  $l_i \lfloor \frac{\Delta t}{W_i} \rfloor$  is the minimum possible jobs,  $n_i^{\min}$ , and  $h_i (\lceil \frac{\Delta t}{W_i} \rceil + 1)$  is the maximum possible jobs,  $n_i^{\max}$ , within  $\Delta t$  under UAM. The AUR can be rewritten as:

$$AUR = \frac{n_1 U_1(s_1^{lf})}{\sum_{i=1}^N n_i U_i(0)} + \dots + \frac{n_N U_N(s_N^{lf})}{\sum_{i=1}^N n_i U_i(0)}.$$

Each  $n_i U_i(s_i^{lf}) / \sum_{i=1}^N n_i U_i(0)$  is a increasing function of  $n_i$  when  $n_i > 0$ , and therefore, the maximum value of each  $n_i$  yields the maximum AUR and the minimum value of each  $n_i$  vice versa.

Thus,

$$\frac{n_i^{min} U_i(s_i^{lf})}{\sum_{i=1}^N n_i^{min} U_i(0)} \leq \frac{n_i U_i(s_i^{lf})}{\sum_{i=1}^N n_i U_i(0)} \leq \frac{n_i^{max} U_i(s_i^{lf})}{\sum_{i=1}^N n_i^{max} U_i(0)}.$$

When a long time interval  $\Delta t$  is considered, since  $n_i^{max} = h_i(\lceil \frac{\Delta t}{W_i} \rceil + 1) < h_i(\frac{\Delta t}{W_i} + 2)$ ,

$$AUR < \frac{\sum_{i=1}^N \frac{h_i}{W_i} U_i(s_i^{lf})}{\sum_{i=1}^N \frac{h_i}{W_i} U_i(0)}.$$

On the other hand, since  $n_i^{min} = l_i \lfloor \frac{\Delta t}{W_i} \rfloor > l_i(\frac{\Delta t}{W_i} - 1)$ ,

$$\frac{\sum_{i=1}^N \frac{l_i}{W_i} U_i(s_i^{lf})}{\sum_{i=1}^N \frac{l_i}{W_i} U_i(0)} < AUR.$$

The sojourn time  $s_i^{lf}$  is within  $[e_i + sm_i, e_i + sm_i + I_i + R_i]$ . With the non-increasing TUFs, the shortest possible sojourn time  $e_i + sm_i$  yields the maximum AUR and the longest possible sojourn time  $e_i + sm_i + I_i + R_i$  yields the minimum AUR.  $\square$

**Lemma 3.3.5** (AUR of Lock-based RUA). *Let the  $i^{th}$  task's jobs arrive under the UAM,  $\langle l_i, h_i, W_i \rangle$ , and be scheduled by RUA. If all jobs are feasible and their TUFs are non-increasing, then the AUR of lock-based sharing, AUR, over time converges into:*

$$\frac{\sum_{i=1}^N \frac{l_i}{W_i} U_i(e_i + q \cdot m_i + I_i + B_i)}{\sum_{i=1}^N \frac{l_i}{W_i} U_i(0)} < AUR < \frac{\sum_{i=1}^N \frac{h_i}{W_i} U_i(e_i + q \cdot m_i)}{\sum_{i=1}^N \frac{h_i}{W_i} U_i(0)},$$

where  $U_i(t)$  is task  $i$ 's utility at time  $t$ , and  $N$  is the number of tasks.

*Proof.* See the proof of Lemma 3.3.4 while replacing  $s$ ,  $s_i^{lf}$ , and  $R_i$  with  $q$ ,  $s_i^{lb}$ , and  $B_i$  respectively, where  $s_i^{lb}$  denotes the sojourn time of  $T_i$  under lock-based sharing.  $\square$

Our comparison of lock-free RUA is against lock-based RUA without nested critical sections, since nested critical sections will likely increase the cost of lock-free RUA, but not that of lock-based RUA.

The AUR bounds of lock-free and lock-based DASA are straightforward to understand. It is computed by considering the constant utility of each task's step-shaped TUF in Lemma 3.3.4 and 3.3.5.

**Corollary 3.3.6** (AUR of Lock-free and Lock-based DASA). *Let the  $i^{\text{th}}$  task's jobs arrive under the UAM,  $\langle l_i, h_i, W_i \rangle$ , and be scheduled by DASA. If all jobs are feasible and their TUFs are non-increasing, then the AUR of lock-based sharing,  $AUR$ , over time converges into one.*

### 3.3.4 Implementation Experience

We implemented lock-based and lock-free objects with RUA in the *meta-scheduler* scheduling framework [52], which allows middleware-level real-time scheduling atop POSIX RTOSes. Our motivation for implementation is to verify our analytical results. We used QNX Neutrino 6.3 RTOS running on a 500MHz, Pentium-III processor in our implementation, which provides an atomic memory-modification operation, the CAS (Compare-And-Swap) instruction. In our study, we used queues, one of the common shared objects, to validate our theorems. We used the lock-free queues introduced in [58] in our implementation.

#### Object Access Times, Critical Load

As Theorem 3.3.3 shows, the tradeoff between lock-based and lock-free sharing under RUA depends upon the time needed for object access—i.e., lock-based object access time  $q$ , and lock-free object access time  $s$ . We measure  $q$  and  $s$  with a 10 task set. Each measurement is an average of approximately 2000 samples.

Figure 3.20 shows  $q$  and  $s$  under increasing number of shared objects accessed by jobs, not allowing any nested critical section.<sup>6</sup> From the figure, we observe that  $q$  is significantly larger than  $s$ . As the number of shared objects accessed by each job,  $q$  is increasing. Note that  $q$  includes the time for lock-based RUA's resource sharing mechanism.

When  $s \ll q$ , Theorem 3.3.3 implies that it increases the likelihood of satisfying the sufficient condition under which lock-free is more likely to perform better than lock-based.

<sup>6</sup>The error bar around each data point represents 95% confidence interval of that data point.

**Figure 3.20:** Object Access Time

The impact of  $q$  and  $s$  on lock-based RUA's and lock-free RUA's performance, respectively, can be measured by evaluating the load at which the schedulers miss task critical times. We measure it using a metric called *Critical time-Miss Load* (CML). The CML of a scheduler is defined as the approximate load *after which* the scheduler begins to miss task critical times. We define approximate load as  $AL = \sum_{i=1}^n e_i/D_i$ , where  $e_i$  is the task computation time excluding object access time, and  $D_i$  is the task critical time.

We exclude object access time for ideal CML, because an ideal implementation of objects for synchronization must have negligibly small – almost zero — object access time. If so, the implementation is ideal in the sense that the scheduler's performance with the (ideal) implementation is the same as that under no object sharing. We call it, the ideal RUA.

We consider a task set of 10 tasks, accessing 10 shared queues, and measure the CML of lock-free, lock-based, and ideal RUA under increasing average job execution times. Figure 3.21 shows the results. We observe that lock-free RUA yields almost the same CML as that of ideal RUA, as it exploits the eliminated blocking times and achieves almost the same performance of RUA without object sharing. Note that the ideal queue and RUA achieve the CML of 1, only at  $\approx 10\text{usec}$  of average execution time. This is because of the algorithm's overhead for scheduling. (RUA's CML of 1, like other scheduling algorithms, is valid at zero job execution times under the assumption of no system overheads, which is not true in practice.)

**Figure 3.21:** Critical Time Miss Load

On the other hand, lock-based RUA's CML converges to 1, only at  $\approx 1$  millisecond. This is precisely because of lock-based RUA's complex operations for resolving jobs' contention for object locks and consequent higher overhead, as manifested by its higher asymptotic complexity and higher object access times in Figure 3.20.

### Accrued Utility, Critical Time-Meets

We now measure the AUR and the CMR of lock-free RUA and lock-based RUA for average job execution times in the range of  $30\text{usec} - 1000\text{usec}$ . CMR is the ratio of the number of tasks that meet their critical times to the total number of task releases. We consider a task set of 10 tasks, accessing 10 shared queues, arbitrarily. Each experiment is repeated to obtain AUR and CMR averages from more than 5000 task arrivals. We consider two classes of TUF shapes in this study: (1) a homogeneous class including just step shapes and (2) a heterogenous class including step, parabolic, and linearly-decreasing shapes.

Figures 3.22 and 3.23 show the AUR and CMR of lock-based and lock-free RUA for step TUFs and heterogenous TUFs, respectively, during under-loads ( $AL = \approx 0.4$ ), under increasing number of shared objects. Figures 3.24 and 3.25 show similar results during overloads ( $AL = \approx 1.1$ ).<sup>7</sup>

<sup>7</sup>In all figures, the error bar around each data point represents 95% confidence interval of that data point.

(a) AUR (b) CMR

**Figure 3.22:** AUR/CMR During Underload, Step TUFs

(a) AUR (b) CMR

**Figure 3.23:** AUR/CMR During Underload, Heterogeneous TUFs

As expected, the figures show that lock-based RUA's AUR and CMR sharply decreases, eventually reaching 0% during overloads, as the number of objects increases. This is because, as the number of objects increases, greater number of task blockings' occurs, due to the large  $r$ , resulting in increased sojourn times, critical time-misses, and consequent abortions.

The performance of lock-free RUA, on the contrary, does not degrade as the number of objects increases. During under-loads, lock-free RUA achieves almost 100% AUR and CMR, whereas during overloads, the algorithm achieves higher AUR by as much as  $\approx 65\%$  and higher CMR, by as much as  $\approx 80\%$  than lock-based. This better performance is directly due to the short  $s$  of

lock-free objects, which results in few retries and thus reduced interferences.

(a) AUR

(b) CMR

**Figure 3.24:** AUR/CMR During Overload, Step TUFs

**Figure 3.25:** AUR/CMR During Overload, Heterogeneous TUFs

Figures 3.25 and 3.26 show lock-based and lock-free RUA's performance, respectively, under heterogeneous TUFs,  $AL = \approx 1.1$ , and increasing number of shared objects. As expected, the figures show that the AUR and CMR of lock-based RUA sharply decreases, eventually reaching 0% during overloads, as the number of objects increases. The performance of lock-free RUA, on the contrary, does not degrade as the number of objects increases.

We repeated similar experiments for increasing number of reader tasks (instead of increasing shared objects) and observed exact similar trends and consistent results in Figure 3.26 (Heterogeneous TUFs,  $AL=0.1-1.1$ ), further illustrating lock-free RUA's superiority over lock-based. We

**Figure 3.26:** AUR/CMR During Increasing Readers, Heterogeneous TUFs

omit more results as they show the same trend and consistency.

## Chapter 4

# Multiprocessor Scheduling I: Underload Scheduling

As discussed in Chapter 1, multiprocessor real-time scheduling is comparatively less studied than single-processor real-time scheduling. The classical hard real-time scheduling objective of meeting all deadlines is a special case of UA scheduling criteria. In this Chapter, we focus on the multiprocessor version of this problem — i.e., scheduling on multiprocessors to meet all deadlines under the periodic arrival model.

The global EDF scheduling algorithm for multiprocessors is a simple extension of the single-processor (optimal) EDF scheduling algorithm<sup>1</sup>. However, global EDF is not optimal for multiprocessors — i.e., all task deadlines may not be satisfied even when the total utilization demand is less than the capacity of all processors. Thus, *feasible* utilization demand bounds — utilization demands below which all deadlines can be satisfied — have been established for global EDF. These include [10, 13, 36]. In [13], Bertogna *et al.* show that each of these bounds do not dominate the other.

---

<sup>1</sup>EDF is optimal for single processors in the sense that EDF can satisfy all deadlines on a single processor when the total utilization demand is less than 1.0 — i.e., during under-loads.

As described in Section 1.3, the Pfair class of algorithms [11, 43] that allow full migration have been shown to achieve a feasible utilization bound that equals the total capacity of all processors, thus they are theoretically optimal. However, Pfair algorithms incur significant run-time overhead due to their quantum-based scheduling approach [31, 69]. Under Pfair, tasks can be decomposed into several small uniform segments, which are then scheduled, causing frequent scheduling and migration. Consequently, the algorithm's optimality is only of theoretical significance. Thus, there does not exist an optimal hard real-time scheduling algorithm for multiprocessors, that does not depend on time quanta.

Since Pfair is only theoretically optimal, algorithms other than Pfair, in particular, global EDF [31, 33, 36, 68], have also been intensively studied, though their feasible utilization bounds are lower. Figure 4.1(a) shows an example task set that global EDF cannot feasibly schedule. Here,  $T_1$  will miss its deadline when the system is given two processors. However, there exists a schedule that meets all task deadlines; this is shown in Figure 4.1(b).

(a)

(b)

**Figure 4.1:** A Task Set that Global EDF Cannot Schedule On Two Processors

Interestingly, we observe in Figure 4.1(b) that the scheduling event at time 1, where task  $T_1$  is split to make all tasks feasible is not a traditional scheduling event (such as a task release or a task completion). This simple observation implies that we may need more scheduling events to split tasks to construct optimal schedules, such as what Pfair's quantum-based approach does. However, it also raises another fundamental question: is it possible to split tasks to construct optimal schedules, not at time quantum, but perhaps at other scheduling events, and consequently avoid Pfair's frequent scheduling and migration overheads? If so, what are those scheduling events?

In this Chapter, we answer these questions. We present an optimal hard real-time scheduling algorithm for multiprocessors, which is *not* based on time quanta. The algorithm called LLREF, is based on the fluid scheduling model and the fairness notion [11]. As compared to global EDF, LLREF provides 100% deadline satisfaction ratio during under-loads, as shown in Figure 4.2. In the figure,  $b_e$  is the feasible utilization demand of global EDF.

(a) global EDF

(b) LLREF

**Figure 4.2:** Scheduling Objective Differences Between global EDF and LLREF

We introduce a novel abstraction for reasoning about task execution behavior on multiprocessors, called the *time and local remaining execution-time plane* (abbreviated as the *T-L plane*). T-L plane makes it possible for us to envision that the entire scheduling over time is just the repetition of T-L planes in various sizes, so that feasible scheduling in a single T-L plane implies feasible scheduling over all times. We define two additional scheduling events and show when they should happen to maintain the fairness of an optimal schedule, and consequently establish LLREF's scheduling optimality. We also show that the overhead of LLREF is tightly bounded, and that bound depends only upon task parameters. Furthermore, our simulation experiments on algorithm overhead validates our analysis.

The rest of the Chapter is organized as follows: Section 4.1 presents LLREF. Section 4.2 establishes LLREF's properties, and reports the simulation results. Some remarks on improving LLREF further are made in Section 4.3.

## 4.1 LLREF Scheduling Algorithm

### 4.1.1 Model

We consider global scheduling, where task migration is not restricted, on an SMP system with  $M$  identical processors. We consider the application to consist of a set of tasks, denoted  $\mathbf{T}=\{T_1, T_2, \dots, T_N\}$ . Tasks are assumed to arrive periodically at their release times  $a_i$ . Each task  $T_i$  has an execution time  $c_i$ , and a relative deadline  $d_i$  which is the same as its period  $p_i$ . The utilization  $u_i$  of a task  $T_i$  is defined as  $c_i/d_i$  and is assumed to be less than 1. Similar to [3, 31], we assume that tasks may be preempted at any time, and are independent, i.e., they do not share resources or have any precedences.

We consider a non-work conserving scheduling policy; thus processors may be idle even when tasks are present in the ready queue. The cost of context switches and task migrations are assumed to be negligible, as in [3, 31].

### 4.1.2 Time and Local Execution Time Plane

In the *fluid* scheduling model, each task executes at a constant rate at all times [44]. The quantum-based Pfair scheduling algorithm, the only known optimal algorithm for the problem that we consider here, is based on the fluid scheduling model, as the algorithm constantly tracks the allocated task execution rate through task utilization. The Pfair algorithm's success in constructing optimal multiprocessor schedules can be attributed to *fairness* — informally, all tasks receive a share of the processor time, and thus are able to simultaneously make progress. P-fairness is a strong notion of fairness, which ensures that at any instant, no application is more than one quantum away from its due share (or fluid schedule) [11, 17].

The significance of the fairness concept on Pfair's optimality is also supported by the fact that task *urgency*, as represented by the task deadline is not sufficient for constructing optimal schedules, as we observe from the poor performance of global EDF for multiprocessors.

**Figure 4.3:** Fluid Schedule versus a Practical Schedule

Toward designing an optimal scheduling algorithm, we thus consider the fluid scheduling model and the fairness notion. To avoid Pfair’s quantum-based approach, we consider an abstraction called the *Time and Local Execution Time Domain Plane* (or abbreviated as the T-L plane), where tokens representing tasks move over time. The T-L plane is inspired by the L-C plane abstraction introduced by Dertouzos *et al.* in [30]. We use the T-L plane to describe fluid schedules, and present a new scheduling algorithm that is able to track the fluid schedule without using time quanta.

Figure 4.3 illustrates the fundamental idea behind the T-L plane. For a task  $T_i$  with  $a_i$ ,  $c_i$  and  $d_i$ , the figure shows a 2-dimensional plane with time represented on the x-axis and the task’s remaining execution time represented on the y-axis. If  $a_i$  is assumed as the origin, the dotted line from  $(0, c_i)$  to  $(d_i, 0)$  indicates the fluid schedule, the slope of which is  $-u_i$ . Since the fluid schedule is ideal but practically impossible, the fairness of a scheduling algorithm depends on how much the algorithm approximates the fluid schedule path.

When  $T_i$  runs like in Figure 4.3, for example, its execution can be represented as a broken line between  $(0, c_i)$  and  $(d_i, 0)$ . Note that task execution is represented as a line whose slope is -1 since x and y axes are in the same scale, and the non-execution over time is represented as a line whose slope is zero. It is clear that the Pfair algorithm can also be represented in the T-L plane as a broken

line based on time quanta.

**Figure 4.4:** T-L Planes

When  $N$  number of tasks are considered, their fluid schedules can be constructed as shown in Figure 4.4, and a right isosceles triangle for all tasks is found between every two consecutive scheduling events. We call this as the T-L plane  $TL^k$ , where  $k$  is simply increasing over time. The size of  $TL^k$  may change over  $k$ . The bottom side of the triangle represents time. The left vertical side of the triangle represents a part of tasks' remaining execution time, which we call the *local remaining execution time*,  $l_i$ , which is supposed to be consumed before each  $TL^k$  ends. Fluid schedules for each task can be constructed as overlapped in each  $TL^k$  plane, while keeping their slopes.

### 4.1.3 Scheduling in T-L planes

The abstraction of T-L planes is significantly meaningful in scheduling for multiprocessors, because T-L planes are repeated over time, and a good scheduling algorithm for a single T-L plane is able to schedule tasks for all repeated T-L planes. Here, good scheduling means being able to con-

struct a schedule that allows all tasks' execution in the T-L plane to approximate the fluid schedule as much as possible. Figure 4.5 details the  $k^{\text{th}}$  T-L plane.

**Figure 4.5:**  $k^{\text{th}}$  T-L Plane

The status of each task is represented as a *token* in the T-L plane. The token's location describes the current time as a value on the horizontal axis and the task's remaining execution time as a value on the vertical axis. The remaining execution time of a task here means one that must be consumed until the time  $t_f^k$ , and not the task's deadline. Hence, we call it, *local* remaining execution time.

As scheduling decisions are made over time, each task's token moves in the T-L plane. Although ideal paths of tokens exist as dotted lines in Figure 4.5, the tokens are only allowed to move in two directions. When the task is selected and executed, the token moves diagonally down, as  $T_N$  moves. Otherwise, it moves horizontally, as  $T_1$  moves. If  $M$  processors are considered, at most  $M$  tokens can diagonally move together. The scheduling objective in the  $k^{\text{th}}$  T-L plane is to make all tokens arrive at the rightmost vertex of the T-L plane—i.e.,  $t_f^k$  with zero local remaining execution time. We call this successful arrival, *locally feasible*. If all tokens are made locally feasible at each T-L plane, they are possible to be scheduled throughout every consecutive T-L planes over time,

approximating all tasks' ideal paths.

For convenience, we define the *local laxity* of a task  $T_i$  as  $t_f^k - t_{cur} - l_i$ , where  $t_{cur}$  is the current time. The oblique side of the T-L plane has an important meaning: when a token hits that side, it implies that the task does not have any local laxity. Thus, if it is not selected immediately, then it cannot satisfy the scheduling objective of local feasibility. We call the oblique side of the T-L plane *no local laxity diagonal* (or NLLD). All tokens are supposed to stay in between the horizontal line and the local laxity diagonal.

We observe that there are two time instants when the scheduling decision has to be made again in the T-L plane. One instant is when the local remaining execution time of a task is completely consumed, and it would be better for the system to run another task instead. When this occurs, the token hits the horizontal line, as  $T_N$  does in Figure 4.5. We call it the *bottom hitting event* (or event B). The other instant is when the local laxity of a task becomes zero so that the task must be selected immediately. When this occurs, the token hits the NLLD, as  $T_1$  does in Figure 4.5. We call it the *ceiling hitting event* (or event C). To distinguish these events from traditional scheduling events such as task releases and task departures, we call events B and C *sub-events*.

To provide local feasibility,  $M$  of the *largest local remaining execution time* tasks are selected first (or LLREF) for every sub-event. We call this, the LLREF scheduling policy. Note that the task having zero local remaining execution time (the token lying on the bottom line in the T-L plane) is not allowed to be selected, which makes our scheduling policy non work-conserving. The tokens for these tasks are called *inactive*, and the others with more than zero local remaining execution time are called *active*. At time  $t_f^k$ , the time instant for the event of the next task release, the next T-L plane  $TL^{k+1}$  starts and LLREF remains valid. Thus, the LLREF scheduling policy is consistently applied for every event.

## 4.2 Algorithm Properties

A fundamental property of the LLREF scheduling algorithm is its scheduling optimality—i.e., the algorithm can meet all task deadlines when the total utilization demand does not exceed the capacity of the processors in the system. In this section, we establish this by proving that LLREF guarantees local feasibility in the T-L plane.

### 4.2.1 Critical Moment

Figure 4.6 shows an example of token flow in the T-L plane. All tokens flow from left to right over time. LLREF selects  $M$  tokens from  $N$  active tokens and they flow diagonally down. The others which are not selected, on the other hand, take horizontal paths. When the event C or B happens, denoted by  $t_j$  where  $0 < j < f$ , LLREF is invoked to make a scheduling decision.

**Figure 4.6:** Example of Token Flow

We define the *local utilization*  $r_{i,j}$  for a task  $T_i$  at time  $t_j$  to be  $\frac{l_{i,j}}{t_f - t_j}$ , which describes how much processor capacity needs to be utilized for executing  $T_i$  within the remaining time until  $t_f$ . Here,  $l_{i,j}$  is the local remaining execution time of task  $T_i$  at time  $t_j$ . When  $k$  is cancelled, it implicitly means the current  $k^{\text{th}}$  T-L plane.

**Theorem 4.2.1** (Initial Local Utilization Value in T-L plane). *Let all tokens arrive at the rightmost*

vertex in the  $(k - 1)^{th}$  T-L plane. Then, the initial local utilization value  $r_{i,0} = u_i$  for all task  $T_i$  in the  $k^{th}$  T-L plane.

*Proof.* If all tokens arrive at  $t_f^{k-1}$  with  $l_i = 0$ , then they can restart in the next T-L plane (the  $k^{th}$  T-L plane) from the positions where their ideal fluid schedule lines start. The slope of the fluid schedule path of task  $T_i$  is  $u_i$ . Thus,  $r_{i,0} = l_{i,0}/t_f = u_i$ .  $\square$

Well-controlled tokens, both departing and arriving points of which are the same as those of their fluid schedule lines in the T-L plane (even though their actual paths in the T-L plane are different from their fluid schedule paths), imply that all tokens are locally feasible. Note that we assume  $u_i \leq 1$  and  $\sum u_i \leq M$ .

#### Figure 4.7: Critical Moment

We define *critical moment* to describe the sufficient and necessary condition that tokens are not locally feasible. (Local infeasibility of the tokens implies that all tokens do not simultaneously arrive at the rightmost vertex of the T-L plane.) Critical moment is the first sub-event time when more than  $M$  tokens simultaneously hit the NLLD. Figure 4.7 shows this. Right after the critical moment, only  $M$  tokens from those on the NLLD are selected. The non-selected ones move out of the triangle, and as a result, they will not arrive at the right vertex of the T-L plane. Note that only horizontal and diagonal moves are permitted for tokens in the T-L plane.

**Theorem 4.2.2** (Critical Moment). *At least one critical moment occurs if and only if tokens are not locally feasible in the T-L plane.*

*Proof.* We prove both the necessary and sufficient conditions.

*Case 1.* Assume that a critical moment occurs. Then, non-selected tokens move out of the T-L plane. Since only -1 and 0 are allowed for the slope of the token paths, it is impossible that the tokens out of the T-L plane reach the rightmost vertex of the T-L plane.

*Case 2.* We assume that when tokens are not locally feasible, no critical moment occurs. If there is no critical moment, then the number of tokens on the NLLD never exceeds  $M$ . Thus, all tokens on the diagonal can be selected by LLREF up to time  $t_f$ . This contradicts our assumption that tokens are not locally feasible.  $\square$

We define *total local utilization* at the  $j^{\text{th}}$  sub-event,  $S_j$ , as  $\sum_i^N r_{i,j}$ .

**Corollary 4.2.3** (Total Local Utilization at Critical Moment). *At the critical moment which is the  $j^{\text{th}}$  sub-event,  $S_j > M$ .*

*Proof.* The local remaining execution time  $l_{i,j}$  for the tasks on the NLLD at the critical moment of the  $j^{\text{th}}$  sub-event is  $t_f - t_j$ , because the T-L plane is an isosceles triangle. Therefore,  $S_j = \sum_{i=1}^N r_{i,j} = \sum_{i=1}^M \frac{t_f - t_j}{t_f - t_j} + \sum_{i=M+1}^N \frac{l_i}{t_f - t_j} > M$ .  $\square$

From the task perspective, the critical moment is the time when more than  $M$  tasks have no local laxity. Thus, the scheduler cannot make them locally feasible with  $M$  processors.

## 4.2.2 Event C

Event C happens when a non-selected token hits the NLLD. Note that selected tokens never hit the NLLD. Event C indicates that the task has no local laxity and hence, should be selected immediately. Figure 4.8 illustrates this, where event C happens at time  $t_c$  when the token  $T_{M+1}$  hits the NLLD.

Note that this is under the basic assumption that there are more than  $M$  tasks, i.e.,  $N > M$ . This implicit assumption holds in Section 4.2.2 and 4.2.3. We will later show the case when  $N \leq M$ .

For convenience, we give lower index  $i$  to the token with higher local utilization, i.e.,  $r_{i,j} \geq r_{i+1,j}$  where  $1 \leq i < N$  and  $\forall j$ , as shown Figure 4.8. Thus, LLREF select tasks from  $T_1$  to  $T_M$  and their tokens move diagonally.

**Figure 4.8:** Event C

**Lemma 4.2.4** (Condition for Event C). *When  $1 - r_{M+1,c-1} \leq r_{M,c-1}$ , the event C occurs at time  $t_c$ , where  $r_{i,c-1} \geq r_{i+1,c-1}$ ,  $1 \leq i < N$ .*

*Proof.* If the sub-event at time  $t_c$  is event C, then token  $T_{M+1}$  must hit the NLLD earlier than when token  $T_M$  hit the bottom of the T-L plane. The time when  $T_{M+1}$  hits the NLLD is  $t_{c-1} + (t_f - t_{c-1} - l_{M+1,c-1})$ . On the contrary, the time when the token  $T_M$  hits the bottom of the T-L plane is  $t_{c-1} + l_{M,c-1}$ .

$$t_{c-1} + (t_f - t_{c-1} - l_{M+1,c-1}) < t_{c-1} + l_{M,c-1}$$

$$1 - \frac{l_{M+1,c-1}}{t_f - t_{c-1}} < \frac{l_{M,c-1}}{t_f - t_{c-1}}.$$

Thus,  $1 - r_{M+1} \leq r_M$  at  $t_{c-1}$ . □

**Corollary 4.2.5** (Necessary Condition for Event C). *Event C occurs at time  $t_c$  only if  $S_{c-1} > M(1 - r_{M+1,c-1})$ , where  $r_{i,c-1} \geq r_{i+1,c-1}$ ,  $1 \leq i < N$ .*

*Proof.*

$$S_{c-1} = \sum_{i=1}^M r_{i,c-1} + \sum_{i=M+1}^N r_{i,c-1} > \sum_{i=1}^M r_{i,c-1} \geq M \cdot r_{M,c-1}.$$

Based on Lemma 4.2.4,  $M \cdot r_{M,c-1} \geq M \cdot (1 - r_{M+1,c-1})$ .  $\square$

**Theorem 4.2.6** (Total Local Utilization for Event C). *When event C occurs at  $t_c$  and  $S_{c-1} \leq M$ , then  $S_c \leq M$  for  $\forall c$  where  $0 < c \leq f$ , and  $r_{i,c-1} \geq r_{i+1,c-1}$ ,  $1 \leq i < N$ .*

*Proof.* We let  $t_c - t_{c-1} = t_f - t_{c-1} - l_{M+1,c-1}$  as  $\delta$ . Total local remaining execution time at  $t_{c-1}$  is  $\sum_{i=1}^N l_{i,c-1} = (t_f - t_{c-1})S_{c-1}$  and it decreases by  $M \times \delta$  at  $t_c$  as  $M$  tokens move diagonally. Therefore,

$$(t_f - t_c)S_c = (t_f - t_{c-1})S_{c-1} - \delta M.$$

Since  $l_{M+1,c-1} = t_f - t_c$ ,

$$l_{M+1,c-1} \times S_c = (t_f - t_{c-1})S_{c-1} - (t_f - t_{c-1} - l_{M+1,c-1})M.$$

Thus,

$$S_c = \frac{1}{r_{M+1,c-1}} S_{c-1} + \left(1 - \frac{1}{r_{M+1,c-1}}\right) M. \quad (4.1)$$

Equation 4.1 is a linear equation as shown in Figure 4.9.

**Figure 4.9:** Linear Equation for Event C

According to Corollary 4.2.5, when event C occurs,  $S_{c-1}$  is more than  $M \cdot (1 - r_{M+1,c-1})$ . Since we also assume  $S_{c-1} \leq M$ ,  $S_c \leq M$ .  $\square$

### 4.2.3 Event B

Event B happens when a selected token hits the bottom side of the T-L plane. Note that non-selected tokens never hit the bottom. Event B indicates that the task has no local remaining execution time so that it would be better to give the processor time to another task for execution.

Event B is illustrated in Figure 4.10, where it happens at time  $t_b$  when the token of  $T_M$  hits the bottom. As we do for the analysis of event C, we give lower index  $i$  to the token with higher local utilization, i.e.,  $r_{i,j} \geq r_{i+1,j}$  where  $1 \leq i < N$ .

**Figure 4.10:** Event B

**Lemma 4.2.7** (Condition for Event B). *When  $1 - r_{M+1,b-1} \geq r_{M,b-1}$ , event B occurs at time  $t_b$ , where  $r_{i,b-1} \geq r_{i+1,b-1}$ ,  $1 \leq i < N$ .*

*Proof.* If the sub-event at time  $t_b$  is event B, then token  $T_M$  must hit the bottom earlier than when token  $T_{M+1}$  hits the NLLD. The time when  $T_M$  hits the bottom and the time when  $T_{M+1}$  hits the NLLD are respectively,  $t_{b-1} + l_{M,b-1}$  and  $t_{b-1} + (t_f - t_{b-1} - l_{M+1,b-1})$ .

$$t_{b-1} + l_{M,b-1} < t_{b-1} + (t_f - t_{b-1} - l_{M+1,b-1}).$$

$$\frac{l_{M,b-1}}{t_f - t_{b-1}} < 1 - \frac{l_{M+1,b-1}}{t_f - t_{b-1}}.$$

Thus,  $r_M \leq 1 - r_{M+1}$  at  $t_{b-1}$ . □

**Corollary 4.2.8** (Necessary Condition for Event B). *Event B occurs at time  $t_b$  only if  $S_{b-1} > M \cdot r_{M,b-1}$ , where  $r_{i,b-1} \geq r_{i+1,b-1}$ ,  $1 \leq i < N$ .*

*Proof.*

$$S_{b-1} = \sum_{i=1}^M r_{i,b-1} + \sum_{i=M+1}^N r_{i,b-1} > \sum_{i=1}^M r_{i,b-1} \geq M \cdot r_{M,b-1}.$$

□

**Theorem 4.2.9** (Total Local Utilization for Event B). *When event B occurs at time  $t_b$  and  $S_{b-1} \leq M$ , then  $S_b \leq M$ , where  $r_{i,b-1} \geq r_{i+1,b-1}$ ,  $1 \leq i < N$ .*

*Proof.*  $t_b - t_{b-1}$  is  $l_{M,b-1}$ . The total local remaining execution time at  $t_{b-1}$  is  $\sum_{i=1}^N l_{i,b-1} = (t_f - t_{b-1})S_{b-1}$ , and this decreases by  $M \cdot l_{M,b-1}$  at  $t_b$  as  $M$  tokens move diagonally. Therefore:

$$(t_f - t_b)S_b = (t_f - t_{b-1})S_{b-1} - M \cdot l_{M,b-1}.$$

Since  $t_f - t_b = t_f - t_{b-1} - l_{M,b-1}$ ,

$$(t_f - t_{b-1} - l_{M,b-1})S_b = (t_f - t_{b-1})S_{b-1} - M \cdot l_{M,b-1}.$$

Thus,

$$S_c = \frac{1}{1 - r_{M,b-1}} S_{c-1} + \left(1 - \frac{r_{M,b-1}}{1 - r_{M,b-1}}\right) M. \quad (4.2)$$

Equation 4.2 is a linear equation as shown in Figure 4.11.

According to Corollary 4.2.8, when event B occurs,  $S_{b-1}$  is more than  $M \cdot r_{M,b-1}$ . Since we also assume  $S_{b-1} \leq M$ ,  $S_b \leq M$ . □

## 4.2.4 Optimality

We now establish LLREF's scheduling optimality by proving its local feasibility in the T-L plane based on our previous results.

In Section 4.2.3 and 4.2.2, we suppose that  $N > M$ . When less than or equal to  $M$  tokens only exist, they are always locally feasible by LLREF in the T-L plane.

**Figure 4.11:** Linear Equation for Event B

**Theorem 4.2.10** (Local Feasibility with Small Number of Tokens). *When  $N \leq M$ , tokens are always locally feasible by LLREF.*

*Proof.* We assume that if  $N \leq M$ , then tokens are not locally feasible by LLREF. If it is not locally feasible, then there should exist at least one critical moment in the T-L plane by Theorem 4.2.2. Critical moment implies at least one non-selected token, which contradicts our assumption since all tokens are selectable.  $\square$

**Figure 4.12:** Token Flow when  $N \leq M$ 

Theorem 4.2.10 is illustrated in Figure 4.12. When the number of tasks is less than the number of processors, LLREF can select all tasks and execute them until their local remaining execution times become zero.

We also observe that at every event B, the number of active tokens is decreasing. In addition, the number of events B in this case is at most  $N$ , since it cannot exceed the number of tokens. Another observation is that event C never happens when  $N \leq M$  since all tokens are selectable and move diagonally.

Now, we discuss the local feasibility when  $N > M$ .

**Theorem 4.2.11** (Local Feasibility with Large Number of Tokens). *When  $N > M$ , tokens are locally feasible by LLREF if  $S_0 < M$ .*

*Proof.* We prove this by induction, based on Theorems 4.2.6 and 4.2.9. Those basically show that if  $S_{j-1} \leq M$ , then  $S_j \leq M$ , where  $j$  is the moment of sub-events. Since we assume that  $S_0 < M$ ,  $S_j$  for all  $j$  never exceeds  $M$  at any sub-event including C and B events. When  $S_j$  is less than  $M$  for all  $j$ , there should be no critical moment in the T-L plane, according to the contraposition of Corollary 4.2.3. By Theorem 4.2.2, no critical moment implies that they are locally feasible.  $\square$

When  $N(> M)$  number of tokens are given in the T-L plane and their  $S_0$  is less than  $M$ , event C and B occur without any critical moment according to Theorem 4.2.11. Whenever event B happens, the number of inactive tokens decreases until there are  $M$  remaining tokens. Then, according to Theorem 4.2.10, all tokens are selectable so that they arrive at the rightmost vertex of the T-L plane with consecutive event B's.

Recall that we consider periodically arriving tasks, and the scheduling objective is to complete all tasks before their deadlines. With continuous T-L planes, if the total utilization of tasks  $\sum_{i=1}^N u_i$  is less than  $M$ , then tokens are locally feasible in the first T-L plane based on Theorems 4.2.10 and 4.2.11. The initial  $S_0$  for the second consecutive T-L plane is less than  $M$  by Theorem 4.2.1 and inductively, LLREF guarantees the local feasibility for every T-L plane, which makes all tasks satisfy their deadlines.

### 4.2.5 Algorithm Overhead

One of the main concerns against global scheduling algorithms (e.g., LLREF, global EDF) is their overhead caused by frequent scheduler invocations. In [69], Srinivasan *et al.* identify three specific overheads:

1. *Scheduling overhead*, which accounts for the time spent by the scheduling algorithm including that for constructing schedules and ready-queue operations;
2. *Context-switching overhead*, which accounts for the time spent in storing the preempted task's context and loading the selected task's context; and
3. *Cache-related preemption delay*, which accounts for the time incurred in recovering from cache misses that a task may suffer when it resumes after a preemption.

Note that when a scheduler is invoked, the context-switching overhead and cache-related preemption delay may not happen. Srinivasan *et al.* also show that the number of task preemptions can be bounded by observing that when a task is scheduled (selected) consecutively for execution, it can be allowed to continue its execution on the same processor. This reduces the number of context-switches and possibility of cache misses. They bound the number of task preemptions under Pfair, illustrating how much a task's execution time inflates due to the aforementioned overhead. They show that, for Pfair, the overhead depends on the time quantum size.

In contrast to Pfair, LLREF is free from time quanta. However, it is clear that LLREF yields more frequent scheduler invocations than global EDF. Note that we use the number of scheduler invocations as a metric for overhead measurement, since it is the scheduler invocation that contributes to all three of the overheads previously discussed. We now derive an upper bound for the scheduler invocations under LLREF.

**Theorem 4.2.12** (Upper-bound on Number of Sub-events in T-L plane). *When tokens are locally feasible in the T-L plane, the number of events in the plane is bounded within  $N + 1$ .*

*Proof.* We consider two possible cases, when a token  $T_i$  hits the NLLD, and it hits the bottom. After  $T_i$  hits the NLLD, it should move along the diagonal to the rightmost vertex of the T-L plane,

because we assume that they are locally feasible. In this case,  $T_i$  raises one sub-event, event C. Note that its final event B at the rightmost vertex occurs together with the next releasing event of another task (i.e., beginning of the new T-L plane). If  $T_i$  hits the bottom, then the token becomes inactive and will arrive at the right vertex after a while. In this case, it raises one sub-event, event B. Therefore, each  $T_i$  can cause one sub-event in the T-L plane. Thus,  $N$  number of tokens can cause  $N + 1$  number of events, which includes  $N$  sub-events and a task release at the rightmost vertex.  $\square$

**Theorem 4.2.13** (Upper-bound of LLREF Scheduler Invocation over Time). *When tasks can be feasibly scheduled by LLREF, the upper-bound on the number of scheduler invocations from time  $t_s$  to  $t_e$  is:*

$$(N + 1) \cdot \left( 1 + \sum_{i=1}^N \left\lceil \frac{t_e - t_s}{p_i} \right\rceil \right),$$

where  $p_i$  is the period of  $T_i$ .

*Proof.* Each T-L plane is constructed between two consecutive events of task release, as shown in Section 4.1.2. The number of task releases during the time between  $t_s$  and  $t_e$  is  $\sum_{i=1}^N \lceil \frac{t_e - t_s}{p_i} \rceil$ . Thus, there can be at most  $1 + \sum_{i=1}^N \lceil \frac{t_e - t_s}{p_i} \rceil$  number of T-L planes between  $t_s$  and  $t_e$ . Since at most  $N + 1$  events can occur in every T-L plane (by Theorem 4.2.12), the upper-bound on the scheduler invocations between  $t_s$  and  $t_e$  is  $(N + 1) \cdot (1 + \sum_{i=1}^N \lceil \frac{t_e - t_s}{p_i} \rceil)$ .  $\square$

Theorem 4.2.13 shows that the number of scheduler invocations of LLREF is primarily dependent on  $N$  and each  $p_i$  — i.e., more number of tasks or more frequent releases of tasks results in increased overhead under LLREF.

## Experimental Evaluation

We conducted simulation-based experimental studies to validate our analytical results on LLREF's overhead. We consider an SMP machine with four processors. We consider four tasks running on the system. Their execution times and periods are given in Table 4.1. The total utilization

is approximately 1.5, which is less than 4, the capacity of processors. Therefore, LLREF can schedule all tasks to meet their deadlines. Note that this task set's  $\alpha$  (i.e.,  $\max^N \{u_i\}$ ) is 0.818, but it does not affect the performance of LLREF, as opposed to that of global EDF.

**Table 4.1:** Task Parameters (4 Task Set)

<i>Tasks</i>	$c_i$	$p_i$	$u_i$
$T_1$	9	11	0.818
$T_2$	5	25	0.2
$T_3$	3	30	0.1
$T_4$	5	14	0.357

**Figure 4.13:** Scheduler Invocation Frequency with 4 Tasks

To evaluate LLREF's overhead in terms of the number of scheduler invocations, we define the scheduler invocation frequency as the number of scheduler invocations during a time interval  $\Delta t$  divided by  $\Delta t$ . We set  $\Delta t$  as 10. According to Theorem 4.2.13, the upper-bound on the number of scheduler invocations in this case is  $(4 + 1) \cdot (1 + \lceil 10/11 \rceil + \lceil 10/25 \rceil + \lceil 10/30 \rceil + \lceil 10/14 \rceil) = 25$ . Therefore, the upper-bound on the scheduler invocation frequency is 2.5.

In Figure 4.13, the upper-bound on the scheduler invocation frequency and the measured frequency are shown as a dotted line and a fluctuating line, respectively. We observe that the actual measured frequency respects the upper-bound.

We now consider eight tasks; the task parameters are given in Table 4.14. The total utilization for this set is approximately 3.72, which is slightly less than the number of processors. Therefore,

**Table 4.2:** Task Parameters (8 Task Set)

<i>Tasks</i>	$c_i$	$p_i$	$u_i$
$T_1$	3	7	0.429
$T_2$	1	16	0.063
$T_3$	5	19	0.263
$T_4$	4	5	0.8
$T_5$	2	26	0.077
$T_6$	15	26	0.577
$T_7$	20	29	0.69
$T_8$	14	17	0.824

**Figure 4.14:** Scheduler Invocation Frequency with 8 Tasks

LLREF again can schedule all tasks to meet their deadlines. When we set  $\Delta t$  as 10, the upper-bound on the number of scheduler invocations is computed as 88, and the upper-bound on the scheduler invocation frequency is computed as 8.8.

Figure 4.14 shows the upper-bound on the invocation frequency and the actual frequency for the 8-task set. Consistently with the previous case, the actual frequency never moves beyond the upper-bound. We also observe that the average invocation frequencies of the two cases are approximately 1.0 and 4.0, respectively. As expected (by Theorem 4.2.13), the number of tasks proportionally affects LLREF's overhead.

Thus, our experimental results validate our analytical results on LLREF's overhead.

### 4.3 Observations

We have shown that LLREF guarantees local feasibility in the T-L plane. This result can be intuitively understood in that, the algorithm first selects tokens which appear to be going out of the T-L plane because they are closer to the NLLD.

However, we perceive that there could be other possibilities. Theorems 4.2.6 and 4.2.9 are fundamental steps toward establishing the local feasibility in the T-L plane. We observe that the two theorems' proofs are directly associated with the LLREF policy in two ways: one is that they depend on the critical moment, and the other is that the range allowed for  $S_{j-1}$  is computed under the assumption of LLREF, where  $j$  is  $c$  or  $b$ . This implies that scheduling policies other than LLREF may also provide local feasibility, as long as they maintain the critical moment concept. (The range allowed for  $S_{j-1}$  could be recomputed for each different scheduling event.) The rules of such scheduling policies can include: (1) selecting as many active tokens as possible up to  $M$  at every sub-event; and (2) including all tokens on the NLLD for selection at every sub-event.

If such algorithms can be designed, then tradeoffs can be established between them and LLREF — e.g., on algorithm complexity, frequency of event C, etc. Note that even in such cases, the upper-bound of scheduler invocations given in Theorems 4.2.12 and 4.2.13 hold, because those theorems are not derived under the assumption of LLREF. Designing such algorithms is a topic for further study.

We also regard that each task's scheduling parameters including execution times and periods are sufficiently longer than the system clock tick, so that they can be assumed as floating point numbers rather than integers. This assumption may not be sufficient for the case when task execution time is shorter so that integer execution times are a better assumption. However, our intuition strongly suggests that even when task parameters are considered as integers, it should be possible to extend our algorithm to achieve a certain level of optimality—e.g., the error between each task's deadline and its completion time is bounded within a finite number of clock ticks.

## Chapter 5

# Multiprocessor Scheduling II: Overload and UA Scheduling

In this Chapter, we build upon LLREF and present a UA scheduling algorithm that maximizes total accrued utility, called the *global Multiprocessor Utility Accrual scheduling algorithm* (or gMUA). We consider the problem of global UA scheduling on an SMP system with  $M$  identical processors. LLREF, as well as global EDF, do not consider non-step TUF time constraints and overloads. In contrast, gMUA considers both, and provides the same feasible utilization demand bound as that of global EDF. This is illustrated in Figure 5.1.

(a) global EDF

(b) gMUA

**Figure 5.1:** Scheduling Objective Differences Between gMUA and Global EDF

gMUA considers repeatedly occurring application activities that are subject to variable execution

times, step and non-step TUF time constraints, and overloads. To account for uncertainties in activity execution behaviors, we consider a stochastic model, where activity execution demand is stochastically expressed. Activities repeatedly arrive with a known minimum inter-arrival time, which can be violated. For such a model, gMUA's objective is to:

- (1) provide statistical assurances on individual activity timeliness behavior including probabilistically-satisfied lower bounds on each activity's maximum utility; (2) provide assurances on system-level timeliness behavior including assured lower bound on the sum of activities' attained utilities; and (3) maximize the sum of activities' attained utilities.

This problem has not been studied in the past and is  $\mathcal{NP}$ -hard. gMUA is a polynomial-time heuristic algorithm for the problem. We establish several properties of gMUA, including optimal total utility for the special case of step TUFs and when the total utilization demand does not exceed global EDF's utilization bound, conditions under which individual activity utility lower bounds are satisfied, and a lower bound on system-wide total accrued utility. We also show that the algorithm's assurances have bounded sensitivity to variations in execution time demand estimates, in the sense that the assurances hold as long as the variations satisfy a sufficient condition that we present. Further, we show that the algorithm is robust against a variant of the Dhall effect.

The rest of the Chapter is organized as follows: In Section 5.1, we state gMUA's models and objectives. In Section 5.2, we present gMUA, describing its rationale and design. We establish gMUA's properties in Section 5.3. Section 5.4 reports our simulation results.

## 5.1 Models and Objective

### 5.1.1 Activity Model

We consider the application to consist of a set of tasks, denoted  $\mathbf{T}=\{T_1, T_2, \dots, T_N\}$ . Each task  $T_i$  has a number of instances, called jobs, and these jobs may be released either periodically or sporadically with a known minimal inter-arrival time. The  $j^{\text{th}}$  job of task  $T_i$  is denoted as  $J_{i,j}$ .

The period or minimal inter-arrival time of a task  $T_i$  is denoted as  $p_i$ . All tasks are assumed to be independent, i.e., they do not share any resource or have any precedences. The basic scheduling entity that we consider is the job abstraction. Thus, we use  $J$  to denote a job without being task specific, as seen by the scheduler at any scheduling event.

A job's time constraint is specified using a TUF. Jobs of the same task have the same TUF. We use  $U_i()$  to denote the TUF of task  $T_i$ . Thus, completion of the job  $J_{i,j}$  at time  $t$  will yield an utility of  $U_i(t)$ .

TUFs can be classified into unimodal and multimodal functions. Unimodal TUFs are those for which any decrease in utility cannot be followed by an increase. Figure 1.1 shows examples. TUFs which are not unimodal are multimodal. In this section, we focus on *non-increasing* unimodal TUFs, as they encompass majority of the time constraints in our motivating applications.

Each TUF  $U_i$  of  $J_{i,j}$  has an initial time  $t_{i,j}^I$  and a termination time  $t_{i,j}^X$ . Initial and termination times are the earliest and the latest times for which the TUF is defined, respectively. We assume that  $t_{i,j}^I$  is the arrival time of job  $J_{i,j}$ , and  $t_{i,j}^X - t_{i,j}^I$  is the period or minimal inter-arrival time  $p_i$  of the task  $T_i$ . If  $J_{i,j}$ 's  $t_{i,j}^X$  is reached and execution of the corresponding job has not been completed, an exception is raised, and the job is aborted.

### 5.1.2 Job Execution Time Demands

We estimate the statistical properties, e.g., distribution, mean, variance, of job execution time demand rather than the worst-case demand because: (1) applications of interest to us [28, 29] exhibit a large variation in their *actual workload*. Thus, the statistical estimation of the demand is much more stable and hence more predictable than the actual workload; (2) worst-case workload is usually a very conservative prediction of the actual workload [8], resulting in resource over-supply; and(3) allocating execution times based on the statistical estimation of tasks' demands can provide statistical performance assurances, which is sufficient for our motivating applications.

Let  $Y_i$  be the random variable of a task  $T_i$ 's execution time demand. Estimating the execution

time demand distribution of the task involves two steps: (1) profiling its execution time usage, and (2) deriving the probability distribution of that usage. A number of measurement-based, off-line and online profiling mechanisms exist (e.g., [87]). We assume that the mean and variance of  $Y_i$  are finite and determined through either online or off-line profiling.

We denote the *expected* execution time demand of a task  $T_i$  as  $E(Y_i)$ , and the variance on the demand as  $Var(Y_i)$ .

### 5.1.3 Statistical Timeliness Requirement

We consider a task-level statistical timeliness requirement: Each task must accrue some percentage of its maximum possible utility with a certain probability. For a task  $T_i$ , this requirement is specified as  $\{\nu_i, \rho_i\}$ , which implies that  $T_i$  must accrue at least  $\nu_i$  percentage of its maximum possible utility with the probability  $\rho_i$ . This is also the requirement of each job of  $T_i$ . Thus, for example, if  $\{\nu_i, \rho_i\} = \{0.7, 0.93\}$ , then  $T_i$  must accrue at least 70% of its maximum possible utility with a probability no less than 93%. For step TUFs,  $\nu$  can only take the value 0 or 1. Note that the objective of always satisfying all task deadlines is the special case of  $\{\nu_i, \rho_i\} = \{1.0, 1.0\}$ .

This statistical timeliness requirement on the utility of a task implies a corresponding requirement on the range of task sojourn times. Since we focus on non-increasing unimodal TUFs, upper-bounding task sojourn times will lower-bound task utilities.

### 5.1.4 Scheduling Objective

We consider a two-fold scheduling criterion: (1) assure that each task  $T_i$  accrues the specified percentage  $\nu_i$  of its maximum possible utility with at least the specified probability  $\rho_i$ ; and (2) maximize the system-level total attained utility. We also desire to obtain a lower bound on the system-level total attained utility. Also, when it is not possible to satisfy  $\rho_i$  for each task (e.g., due to overloads), our objective is to maximize the system-level total utility.

This problem is  $\mathcal{NP}$ -hard because it subsumes the  $\mathcal{NP}$ -hard problem of scheduling dependent tasks with step TUFs on one processor [27].

## 5.2 The gMUA Algorithm

### 5.2.1 Bounding Accrued Utility

Let  $s_{i,j}$  be the sojourn time of the  $j^{\text{th}}$  job of task  $T_i$ , where the sojourn time is defined as the period from the job's release to its completion. Now, task  $T_i$ 's statistical timeliness requirement can be represented as  $Pr(U_i(s_{i,j}) \geq \nu_i \times U_i^{\text{max}}) \geq \rho_i$ . Since TUFs are assumed to be non-increasing, it is sufficient to have  $Pr(s_{i,j} \leq D_i) \geq \rho_i$ , where  $D_i$  is the upper bound on the sojourn time of task  $T_i$ . We call  $D_i$  "critical time" hereafter, and it is calculated as  $D_i = U_i^{-1}(\nu_i \times U_i^{\text{max}})$ , where  $U_i^{-1}(x)$  denotes the inverse function of TUF  $U_i(\cdot)$ . Thus,  $T_i$  is (probabilistically) assured to accrue at least the utility percentage  $\nu_i = U_i(D_i)/U_i^{\text{max}}$ , with the probability  $\rho_i$ .

Note that the period or minimum inter-arrival time  $p_i$  and the critical time  $D_i$  of the task  $T_i$  have the following relationships: (1)  $p_i = D_i$  for a binary-valued, downward step TUF; and (2)  $p_i \geq D_i$ , for other non-increasing TUFs.

### 5.2.2 Bounding Utility Accrual Probability

Since task execution time demands are stochastically specified (through means and variances), we need to determine the actual execution time that must be allocated to each task, such that the desired utility accrual probability  $\rho_i$  is satisfied. Further, this execution time allocation must account for the uncertainty in the execution time demand specification (i.e., the variance factor).

Given the mean and the variance of a task  $T_i$ 's execution time demand  $Y_i$ , by a one-tailed version of the Chebyshev's inequality, when  $y \geq E(Y_i)$ , we have:

$$Pr[Y_i < y] \geq \frac{(y - E(Y_i))^2}{Var(Y_i) + (y - E(Y_i))^2} \quad (5.1)$$

From a probabilistic point of view, Equation 5.1 is the direct result of the cumulative distribution function of task  $T_i$ 's execution time demands—i.e.,  $F_i(y) = Pr[Y_i \leq y]$ . Recall that each job of task  $T_i$  must accrue  $\nu_i$  percentage of its maximum utility with a probability  $\rho_i$ . To satisfy this requirement, we let  $\rho'_i = \frac{(C_i - E(Y_i))^2}{Var(Y_i) + (C_i - E(Y_i))^2} \geq \rho_i$  and obtain the minimum required execution time  $C_i = E(Y_i) + \sqrt{\frac{\rho'_i \times Var(Y_i)}{1 - \rho'_i}}$ .

Thus, the gMUA algorithm allocates  $C_i$  execution time units to each job  $J_{i,j}$  of task  $T_i$ , so that the probability that job  $J_{i,j}$  requires no more than the allocated  $C_i$  execution time units is at least  $\rho_i$ —i.e.,  $Pr[Y_i < C_i] \geq \rho'_i \geq \rho_i$ . We set  $\rho'_i = (\max\{\rho_i\})^{\frac{1}{n}}$ ,  $\forall i$  to satisfy requirements given by  $\rho_i$ . Supposing that each task is allocated  $C_i$  time within its  $p_i$ , the actual demand of each task often vary. Some jobs of the task may complete its execution before using up its allocated time and the others may not. gMUA probabilistically schedules the jobs of a task  $T_i$  to provide assurance  $\rho'_i$  ( $\geq \rho_i$ ) as long as they are satisfying a certain feasibility test.

**Figure 5.2:** Transformation Array and gMUA

Figure 5.2 shows our method of transforming the stochastic execution time demand ( $E(Y_i)$  and  $Var(Y_i)$ ) into execution time allocation  $C_i$ . Note that this transformation is independent of our proposed scheduling algorithm.

### 5.2.3 Algorithm Description

gMUA's scheduling events include job arrival and job completion. To describe gMUA, we define the following variables and auxiliary functions:

- $\zeta_r$ : current job set in the system including running jobs and unscheduled jobs.
- $\sigma_{tmp}, \sigma_a$ : a temporary schedule;  $\sigma_m$ : schedule for processor  $m$ , where  $m \leq M$ .
- $J_k.C(t)$ :  $J_k$ 's remaining allocated execution time.
- `offlineComputing()` is computed at time  $t = 0$  once. For a task  $T_i$ , it computes  $C_i$  as  $C_i = E(Y_i) + \sqrt{\frac{\rho_i \times Var(Y_i)}{1 - \rho_i}}$ .
- `UpdateRAET( $\zeta_r$ )` updates the remaining allocated execution time of all jobs in the set  $\zeta_r$ .
- `feasible( $\sigma$ )` returns a boolean value denoting schedule  $\sigma$ 's feasibility; `feasible( $J_k$ )` denotes job  $J_k$ 's feasibility. For  $\sigma$  (or  $J_k$ ) to be feasible, the predicted completion time of each job in  $\sigma$  (or  $J_k$ ), must not exceed its critical time.
- `sortByECF( $\sigma$ )` sorts jobs of  $\sigma$  in the order of earliest critical time first.
- `findProcessor()` returns the ID of the processor on which the currently assigned tasks have the shortest sum of allocated execution times.
- `append( $J_k, \sigma$ )` appends job  $J_k$  at rear of schedule  $\sigma$ .
- `remove( $J_k, \sigma$ )` removes job  $J_k$  from schedule  $\sigma$ .
- `removeLeastPUDJob( $\sigma$ )` removes job with the least *potential utility density* (or PUD) from schedule  $\sigma$ . PUD is the ratio of the expected job utility (obtained when job is immediately executed to completion) to the remaining job allocated execution time, i.e., PUD of a job  $J_k$  is  $\frac{U_k(t+J_k.C(t))}{J_k.C(t)}$ . Thus, PUD measures the job's "return on investment." Function returns the removed job.
- `headOf( $\sigma_m$ )` returns the set of jobs that are at the head of schedule  $\sigma_m$ ,  $1 \leq m \leq M$ .

A description of gMUA at a high level of abstraction is shown in Algorithm 5. The procedure `offlineComputing()` is included in line 4, although it is executed only once at  $t = 0$ . When gMUA is invoked, it updates the remaining allocated execution time of each job. The remaining allocated execution times of running jobs are decreasing, while those of unscheduled jobs remain

**Algorithm 5:** gMUA

---

```

1 Input   :  $\mathbf{T}=\{T_1,\dots,T_N\}$ ,  $\zeta_r=\{J_1,\dots,J_n\}$ ,  $M$ :# of processors
2 Output : array of dispatched jobs to processor  $p$ ,  $Job_p$ 
3 Data:  $\{\sigma_1, \dots, \sigma_M\}$ ,  $\sigma_{tmp}$ ,  $\sigma_a$ 
4 offlineComputing( $\mathbf{T}$ );
5 Initialization:  $\{\sigma_1, \dots, \sigma_M\} = \{0, \dots, 0\}$ ;
6 UpdateRAET( $\zeta_r$ );
7 for  $\forall J_k \in \zeta_r$  do
8    $J_k.PUD = \frac{U_k(t+J_k.C(t))}{J_k.C(t)}$ ;
9  $\sigma_{tmp} = \text{sortByECF}(\zeta_r)$ ;
10 for  $\forall J_k \in \sigma_{tmp}$  from head to tail do
11   if  $J_k.PUD > 0$  then
12      $m = \text{findProcessor}()$ ;
13      $\text{append}(J_k, \sigma_m)$ ;
14 for  $m = 1$  to  $M$  do
15    $\sigma_a = \text{null}$ ;
16   while  $!feasible(\sigma_m)$  and  $!IsEmpty(\sigma_m)$  do
17      $t = \text{removeLeastPUDJob}(\sigma_m)$ ;
18      $\text{append}(t, \sigma_a)$ ;
19    $\text{sortByECF}(\sigma_a)$ ;
20    $\sigma_m += \sigma_a$ ;
21  $\{Job_1, \dots, Job_M\} = \text{headOf}(\{\sigma_1, \dots, \sigma_M\})$ ;
22 return  $\{Job_1, \dots, Job_M\}$ ;

```

---

constant. The algorithm then computes the PUDs of all jobs.

The jobs are then sorted in the order of earliest critical time first (or ECF), in line 9. In each step of the for loop from line 10 to line 13, the job with the earliest critical time is selected to be assigned to a processor. The processor that yields the shortest sum of allocated execution times of all jobs in its local schedule is selected for assignment (procedure `findProcessor()`). The rationale for this choice is that the shortest summed execution time processor results in the nearest scheduling event after assigning each job, which is to establish the same schedule as the global EDF does. Then, the job  $J_k$  with the earliest critical time is inserted into the local schedule  $\sigma_m$  of the selected processor  $m$ .

In the for-loop from line 14 to line 20, gMUA attempts to make each local schedule feasible by removing the lowest PUD job. In line 16, if  $\sigma_m$  is not feasible, then gMUA removes the job with the least PUD from  $\sigma_m$  until  $\sigma_m$  becomes feasible. All removed jobs are temporarily stored in a

schedule  $\sigma_a$  and then appended to each  $\sigma_m$  in ECF order. Note that simply aborting the removed jobs may result in decreased accrued utility. This is because, the algorithm may decide to remove a job which is estimated to have a longer allocated execution time than its actual one, even though it may be able to accrue utility. For this case, gMUA gives the job another chance to be scheduled instead of aborting it, which eventually makes the algorithm more robust. Finally, each job at the head of  $\sigma_m, 1 \leq m \leq M$  is selected for execution on the respective processor.

gMUA's time cost depends upon that of procedures `sortByECF()`, `findprocessor()`, `append()`, `feasible()`, and `removeLeastPUDJob()`. With  $n$  jobs, `sortByECF()` costs  $O(n \log n)$ . For SMPs with restricted number of processors, `findprocessor()`'s costs  $O(M)$ . While `append()` costs  $O(1)$  time, both `feasible()` and `removeLeastPUDJob()` costs  $O(n)$ . The while-loop in line 16 iterates at most  $n$  times, costing the entire loop  $O(n^2)$ . Thus, the algorithm costs  $O(Mn^2)$ . However,  $M$  of SMPs is usually small (e.g., 16) and bounded with respect to the problem size of number of tasks. Thus, gMUA costs  $O(n^2)$ .

gMUA's  $O(n^2)$  cost is similar to that of many past UA algorithms [64]. Our prior implementation experience with UA scheduling at the middleware-level have shown that the overheads are in the magnitude of sub-milliseconds [52] (sub-microsecond overheads may be possible at the kernel-level). We anticipate a similar overhead magnitude for gMUA. Though this cost is higher than that of many traditional algorithms, the cost is justified for applications with longer execution time magnitudes (e.g., milliseconds to minutes) such as those that we focus here. Of course, this high cost cannot be justified for every application.<sup>1</sup>

---

<sup>1</sup>When UA scheduling is desired with low overhead, solutions and tradeoffs exist. Examples include linear-time stochastic UA scheduling [51], and using special-purpose hardware accelerators for UA scheduling (analogous to floating-point co-processors) [61].

## 5.3 Algorithm Properties

### 5.3.1 Timeliness Assurances

We establish gMUA's timeliness assurances under the conditions of (1) independent tasks that arrive periodically, and (2) task utilization demand satisfies any of the feasible utilization bounds for global EDF (GFB, BAK, BCL) in [13].

**Theorem 5.3.1** (Optimal Performance with Step Shaped TUFs). *Suppose that only step shaped TUFs are allowed under conditions (1) and (2). Then, a schedule produced by global EDF is also produced by gMUA, yielding equal total utilities. This is a critical time-ordered schedule.*

*Proof.* We prove this by examining Algorithm 5. In line 9, the queue  $\sigma_{tmp}$  is sorted in a non-decreasing critical time order. In line 12, the function `findProcessor()` returns the index of the processor on which the summed execution time of assigned tasks is the shortest among all processors. Assume that there are  $N$  tasks in the current ready queue. We consider two cases: (1)  $N \leq M$  and (2)  $N > M$ . When  $N \leq M$ , the result is trivial — gMUA's schedule of tasks on each processor is identical to that produced by EDF (every processor has a single task or none assigned). When  $N > M$ , task  $T_i$  ( $M < i \leq N$ ) will be assigned to the processor whose tasks have the shortest summed execution time. This implies that this processor will have the earliest completion for all assigned tasks up to  $T_{i-1}$ , so that the event that will assign  $T_i$  will occur by this completion. Note that tasks in  $\sigma_{tmp}$  are selected to be assigned to processors according to ECF. This is precisely the global EDF schedule, as we consider a critical time of UA scheduling as a deadline of traditional hard real-time scheduling. Under conditions (1) and (2), EDF meets all deadlines. Thus, each processor always has a feasible schedule, and the while-block from line 16 to line 18 will never be executed. Thus, gMUA produces the same schedule as global EDF.  $\square$

Some important corollaries about gMUA's timeliness behavior can be deduced from EDF's behavior under conditions (1) and (2).

**Corollary 5.3.2.** *Under conditions (1) and (2), gMUA always completes the allocated execution time of all tasks before their critical times.*

**Theorem 5.3.3** (Statistical Task-Level Assurance). *Under conditions (1) and (2), gMUA meets the statistical timeliness requirement  $\{\nu_i, \rho_i\}$  for each task  $T_i$ .*

*Proof.* From Corollary 5.3.2, all allocated execution times of tasks can be completed before their critical times. Further, based on the results of Equation 5.1, among the actual processor time of task  $T_i$ 's jobs, at least  $\rho_i$  of them have lesser actual execution time than the allocated execution time. Thus, gMUA can satisfy at least  $\rho_i$  critical times—i.e., the algorithm accrues  $\nu_i$  utility with a probability of at least  $\rho_i$ .  $\square$

**Theorem 5.3.4** (System-Level Utility Assurance). *Under conditions (1) and (2), if a task  $T_i$ 's TUF has the highest height  $U_i^{max}$ , then the system-level utility ratio, defined as the utility accrued by gMUA with respect to the system's maximum possible utility, is at least  $\frac{\rho_1 \nu_1 U_1^{max}/p_1 + \dots + \rho_N \nu_N U_N^{max}/p_N}{U_1^{max}/p_1 + \dots + U_N^{max}/p_N}$ .*

*Proof.* We denote the number of jobs released by task  $T_i$  as  $m_i$ . Each  $m_i$  is computed as  $\frac{\Delta t}{p_i}$ , where  $\Delta t$  is a time interval. Task  $T_i$  can accrue at least  $\nu_i$  percentage of its maximum possible utility with the probability  $\rho_i$ . Thus, the ratio of the system-level accrued utility to the system's maximum possible utility is  $\frac{\rho_1 \nu_1 U_1^{max} m_1 + \dots + \rho_N \nu_N U_N^{max} m_N}{U_1^{max} m_1 + \dots + U_N^{max} m_N}$ . Thus, the formula comes to  $\frac{\rho_1 \nu_1 U_1^{max}/p_1 + \dots + \rho_N \nu_N U_N^{max}/p_N}{U_1^{max}/p_1 + \dots + U_N^{max}/p_N}$ .  $\square$

### 5.3.2 Dhall Effect

The *Dhall effect* [33] shows that there exists a task set that requires nearly 1 total utilization demand, but cannot be scheduled to meet all deadlines under global EDF and RM even with infinite number of processors. Prior research has revealed that this is caused by the poor performance of global EDF and RM when the task set contains both high utilization tasks and low utilization tasks together. This phenomena, in general, can also affect UA scheduling algorithms' performance, and counter such algorithms' ability to maximize the total attained utility. We discuss this with an

example inspired from [86]. We consider the case when the execution time demands of all tasks are constant with no variance, and gMUA estimates them accurately.

**Example A.** Consider  $M + 1$  periodic tasks that are scheduled on  $M$  processors under global EDF. Let task  $T_i$ , where  $1 \leq i \leq M$ , have  $p_i = D_i = 1$ ,  $C_i = 2\epsilon$ , and task  $T_{M+1}$  have  $p_{M+1} = D_{M+1} = 1 + \epsilon$ ,  $C_{M+1} = 1$ . We assume that each task  $T_i$  has a step shaped TUF with height  $U_i^{max}$  and task  $T_{M+1}$  has a step shaped TUF with height  $U_{M+1}^{max}$ . When all tasks arrive at the same time, tasks  $T_i$  will execute immediately and complete their execution  $2\epsilon$  time units later. Task  $T_{M+1}$  then executes from time  $2\epsilon$  to time  $1 + 2\epsilon$ . Since task  $T_{M+1}$ 's critical time — we assume here it is the same as its period — is  $1 + \epsilon$ , it begins to miss its critical time. By letting  $M \rightarrow \infty$ ,  $\epsilon \rightarrow 0$ ,  $U_i^{max} \rightarrow 0$  and  $U_{M+1}^{max} \rightarrow \infty$ , we have a task set, whose total utilization demand is near 1 and the maximum possible total attained utility is infinite, but that finally accrues zero total utility even with infinite number of processors.

We call this phenomena as the *UA Dhall effect*. Conclusively, one of the reasons why global EDF is inappropriate as a UA scheduler is that it is prone to suffer this effect. However, gMUA overcomes this phenomena.

**Example B.** Consider the same scenario as in Example A, but now, let the task set be scheduled by gMUA. In Algorithm 5, gMUA first tries to schedule tasks like global EDF, but it will fail to do so as we saw in Example A. When gMUA finds that  $T_{M+1}$  will miss its critical time on processor  $m$  (where  $1 \leq m \leq M$ ), the algorithm will select a task with lower PUD on processor  $m$  for removal. On processor  $m$ , there should be two tasks,  $T_m$  and  $T_{M+1}$ .  $T_m$  is one of  $T_i$  where  $1 \leq i \leq M$ . When letting  $U_i^{max} \rightarrow 0$  and  $U_{M+1}^{max} \rightarrow \infty$ , the PUD of task  $T_m$  is almost zero and that of task  $T_{M+1}$  is infinite. Therefore, gMUA removes  $T_m$  and eventually accrues infinite utility as expected.

Under the case when *Dhall effect* occurs, we can establish *UA Dhall effect* by assigning extremely high utility to the task that will be selected and miss deadline by global EDF. It also implies that the scheduling algorithm suffering from *Dhall effect* will likely suffer from *UA Dhall effect*, when it schedules the tasks that are subject to TUF time constraints.

The fact that gMUA is more robust against *UA Dhall effect* than global EDF can be observed in

our simulation experiments (see Section 5.4).

### 5.3.3 Sensitivity of Assurances

gMUA is designed under the assumption that task expected execution time demands and the variances on the demands — i.e., the algorithm inputs  $E(Y_i)$  and  $Var(Y_i)$  — are correct. However, it is possible that these inputs may have been miscalculated (e.g., due to errors in application profiling) or that the input values may change over time (e.g., due to changes in application's execution context). To understand gMUA's behavior when this happens, we assume that the expected execution time demands,  $E(Y_i)$ 's, and their variances,  $Var(Y_i)$ 's, are erroneous, and develop the sufficient condition under which the algorithm is still able to meet  $\{\nu_i, \rho_i\}$  for all tasks  $T_i$ .

Let a task  $T_i$ 's correct expected execution time demand be  $E(Y_i)$  and its correct variance be  $Var(Y_i)$ , and let an erroneous expected demand  $E'(Y_i)$  and an erroneous variance  $Var'(Y_i)$  be specified as the input to gMUA. Let the task's statistical timeliness requirement be  $\{\nu_i, \rho_i\}$ . We show that if gMUA can satisfy  $\{\nu_i, \rho_i\}$  with the correct expectation  $E(Y_i)$  and the correct variance  $Var(Y_i)$ , then there exists a sufficient condition under which the algorithm can still satisfy  $\{\nu_i, \rho_i\}$  even with the incorrect expectation  $E'(Y_i)$  and incorrect variance  $Var'(Y_i)$ .

**Theorem 5.3.5.** *Assume that gMUA satisfies  $\{\nu_i, \rho_i\}, \forall i$ , under correct, expected execution time demand estimates,  $E(Y_i)$ 's, and their correct variances,  $Var(Y_i)$ 's. When incorrect expected values,  $E'(Y_i)$ 's, and variances,  $Var'(Y_i)$ 's, are given as inputs instead of  $E(Y_i)$ 's and  $Var(Y_i)$ 's, gMUA satisfies  $\{\nu_i, \rho_i\}, \forall i$ , if  $E'(Y_i) + (C_i - E(Y_i))\sqrt{\frac{Var'(Y_i)}{Var(Y_i)}} \geq C_i, \forall i$ , and the task execution time allocations, computed using  $E'(Y_i)$ 's and  $Var'(Y_i)$ , satisfy any of the feasible utilization bounds for global EDF.*

*Proof.* We assume that if gMUA has correct  $E(Y_i)$ 's and  $Var(Y_i)$ 's as inputs, then it satisfies  $\{\nu_i, \rho_i\}, \forall i$ . This implies that the  $C_i$ 's determined by Equation 5.1 are feasibly scheduled by gMUA satisfying all task critical times:

$$\rho_i = \frac{(C_i - E(Y_i))^2}{Var(Y_i) + (C_i - E(Y_i))^2}. \quad (5.2)$$

However, gMUA has incorrect inputs,  $E'(Y_i)$ 's and  $Var'(Y_i)$ , and based on those, it determines  $C'_i$ s by Equation 5.1 to obtain the probability  $\rho_i, \forall i$ :

$$\rho_i = \frac{(C'_i - E'(Y_i))^2}{Var'(Y_i) + (C'_i - E'(Y_i))^2}. \quad (5.3)$$

Unfortunately,  $C'_i$  that is calculated from the erroneous  $E'(Y_i)$  and  $Var'(Y_i)$  leads gMUA to another probability  $\rho'_i$  by Equation 5.1. Thus, although we expect assurance with the probability  $\rho_i$ , we can only obtain assurance with the probability  $\rho'_i$  because of the error.  $\rho'$  is given by:

$$\rho'_i = \frac{(C'_i - E(Y_i))^2}{Var(Y_i) + (C'_i - E(Y_i))^2}. \quad (5.4)$$

Note that we also assume that tasks with  $C'_i$  satisfy the global EDF's utilization bound; otherwise gMUA cannot provide the assurances. To satisfy  $\{\nu_i, \rho_i\}, \forall i$ , the actual probability  $\rho'_i$  must be greater than the desired probability  $\rho_i$ . Since  $\rho'_i \geq \rho_i$ ,

$$\frac{(C'_i - E(Y_i))^2}{Var(Y_i) + (C'_i - E(Y_i))^2} \geq \frac{(C_i - E(Y_i))^2}{Var(Y_i) + (C_i - E(Y_i))^2}.$$

Hence,  $C' \geq C_i$ . From Equations 5.2 and 5.3,

$$C'_i = E'(Y_i) + (C_i - E(Y_i)) \sqrt{\frac{Var'(Y_i)}{Var(Y_i)}} \geq C_i. \quad (5.5)$$

□

**Corollary 5.3.6.** *Assume that gMUA satisfies  $\{\nu_i, \rho_i\}, \forall i$ , under correct, expected execution time demand estimates,  $E(Y_i)$ 's, and their correct variances,  $Var(Y_i)$ 's. When incorrect expected values,  $E'(Y_i)$ 's, are given as inputs instead of  $E(Y_i)$ 's but with correct variances  $Var(Y_i)$ , gMUA satisfies  $\{\nu_i, \rho_i\}, \forall i$ , if  $E'(Y_i) \geq E(Y_i), \forall i$ , and the task execution time allocations, computed using  $E'(Y_i)$ 's, satisfy the feasible utilization bound for global EDF.*

*Proof.* This can be proved by replacing  $Var'(Y_i)$  with  $Var(Y_i)$  in Equation 5.5. □

Corollary 5.3.6, a special case of Theorem 5.3.5, is intuitively straightforward: It essentially states that if overestimated demands are feasible, then gMUA can still satisfy  $\{\nu_i, \rho_i\}, \forall i$ . Thus, it is desirable to specify larger  $E'(Y_i)$ s as input to the algorithm when there is the possibility of errors in determining the expected demands, or when the expected demands may vary with time.

**Corollary 5.3.7.** *Assume that gMUA satisfies  $\{\nu_i, \rho_i\}, \forall i$ , under correct, expected execution time demand estimates,  $E(Y_i)$ 's, and their correct variances,  $Var(Y_i)$ 's. When incorrect variances,  $Var'(Y_i)$ 's, are given as inputs instead of correct  $Var(Y_i)$ 's but with correct expectations  $E(Y_i)$ 's, gMUA satisfies  $\{\nu_i, \rho_i\}, \forall i$ , if  $Var'(Y_i) \geq Var(Y_i), \forall i$ , and the task execution time allocations, computed using  $E'(Y_i)$ 's, satisfy the feasible utilization bound for global EDF.*

*Proof.* This can be proved by replacing  $E'(Y_i)$  with  $E(Y_i)$  in Equation 5.5. □

## 5.4 Experimental Evaluation

We conducted simulation-based experimental studies to validate our analytical results and to compare gMUA's performance with global EDF. We consider two cases: (1) the demand of all tasks are constant (i.e., no variance) and gMUA exactly estimates the execution time allocation, and (2) the demand of all tasks statistically varies and gMUA probabilistically estimates the execution time allocation for each task. The former experiment is conducted to evaluate gMUA's generic performance as opposed to EDF, while the latter is conducted to validate the algorithm's assurances. The experiments focus on the no dependency case (i.e., each task is assumed to be independent of others).

### 5.4.1 Performance with Constant Demand

We consider an SMP machine with 4 processors. A task  $T_i$ 's period  $p_i (= t_i^X)$  and its expected execution time  $E(Y_i)$  are randomly generated in the range  $[1, 30]$  and  $[1, \alpha \cdot p_i]$ , respectively, where  $\alpha$  is defined as  $\max\{\frac{C_i}{p_i} | i = 1, \dots, n\}$  and  $Var(Y_i)$  are zero. According to [36], EDF's feasible utilization bound depends on  $\alpha$  as well as the number of processors. It implies that no matter how many processors the system has, there exists task sets with total utilization demand,  $u$ , close to 1.0, which cannot be scheduled under EDF satisfying all deadlines. Generally, the performance of global schemes tends to decrease when  $\alpha$  increases.

We consider two TUF shape patterns: (1) all tasks have step shaped TUFs, and (2) a heterogeneous TUF class, including step, linearly decreasing and parabolic shapes. Each TUF's height is randomly generated in the range  $[1,100]$ .

(a) AUR

(b) CMR

**Figure 5.3:** Performance Under Constant Demand, Step TUFs

The number of tasks are determined depending on the given  $u$  and the  $\alpha$  value. We vary the  $u$  from 3 to 6.5, including the case where it exceeds the number of processors. We set  $\alpha$  to 0.4, 0.7, and 1. For each experiment, more than 1000,000 jobs are released. To see the generic performance of gMUA, we assume  $\{\nu_i, \rho_i\} = \{0, 1\}$ .

Figures 5.3 and 5.4 show the accrued utility ratio (AUR) and critical-time meet ratio (CMR) of gMUA and EDF, respectively, under increasing  $u$  (from 3.0 to 6.5) and for the three  $\alpha$  values. AUR is the ratio of total accrued utility to the total maximum possible utility, and CMR is the ratio of the number of jobs meeting their critical times to the total number of job releases. For a task with a step TUF, its AUR and CMR are identical. But the system-level AUR and CMR can be different due to the mix of different utility of tasks.

When all tasks have step TUFs and the total  $u$  satisfies the global EDF's feasible utilization bound, gMUA performs exactly the same to EDF. This validates Theorem 5.3.1.

EDF's performance drops sharply after  $u = 4.0$  (for step TUFs), which corresponds to the number

(a) AUR

(b) CMR

**Figure 5.4:** Performance Under Constant Demand, Heterogeneous TUFs

of processors in our experiments. This is due to EDF's domino effect (originally identified for single processors) that occurs here, when  $u$  exceeds the number of processors. On the other hand, the performance of gMUA gracefully degrades as  $u$  increases and exceeds 4.0, since gMUA selects favors as many feasible, higher PUD tasks as possible, instead of simply favoring earlier deadline tasks.

Observe that EDF begins to miss deadlines much earlier than when  $u = 4.0$ , as indicated in [13]. Even when  $u < 4.0$ , gMUA outperforms EDF in both AUR and CMR. This is because gMUA is likely to find a feasible or at least better schedule even when EDF cannot find a feasible one, as we have seen in Section 5.3.2.

We also observe that  $\alpha$  affects the AUR and CMR of both EDF and gMUA. Despite this effect, gMUA outperforms EDF for the same  $\alpha$  and  $u$  for the reason that we describe above.

We observe similar and consistent trends for tasks with heterogeneous TUFs in Figure 5.4. The figure shows that gMUA is superior to EDF under heterogeneous TUFs and when  $u$  exceeds the number of processors.

### 5.4.2 Performance with Statistical Demand

We now evaluate gMUA's statistical timeliness assurances. The task settings used in our simulation study are summarized in Table 5.1. The table shows the task periods and the maximum utility (or  $U_{max}$ ) of the TUFs. For each task  $T_i$ 's demand  $Y_i$ , we generate normally distributed execution time demands. Task execution times are changed along with the total  $u$ . We consider both homogeneous and heterogeneous TUF shapes as before.

**Table 5.1:** Task Settings

<i>Task</i>	$p_i$	$U_i^{max}$	$\rho_i$	$E(Y_i)$	$Var(Y_i)$
$T_1$	25	400	0.96	3.15	0.01
$T_2$	28	100	0.96	13.39	0.01
$T_3$	49	20	0.96	18.43	0.01
$T_4$	49	100	0.96	23.91	0.01
$T_5$	41	30	0.96	14.98	0.01
$T_6$	49	400	0.96	24.17	0.01

(a) Task-level

(b) System-level

**Figure 5.5:** Performance Under Statistical Demand, Step TUFs

Figures 5.5(a) shows AUR and CMR of each task under increasing total  $u$  of gMUA. For a task with step TUFs, task-level AUR and CMR are identical, as satisfying the critical time implies the accrual of a constant utility. But the system-level AUR and CMR are different as satisfying the critical time of each task does not always yield the same amount of utility.

(a) Task-level (b) System-level

**Figure 5.6:** Performance Under Statistical Demand, Heterogenous TUFs

Figure 5.5(a) shows that all tasks under gMUA accrue 100% AUR and CMR within the global EDF's bound (i.e.,  $u < \approx 2.5$  here), thus satisfying the desired  $\{\nu_i, \rho_i\} = \{1, 0.96\}, \forall i$ . This validates Theorem 5.3.3.

Under the condition beyond what Theorem 5.3.3 indicates, gMUA achieves graceful performance degradation in both AUR and CMR in Figure 5.5(b), as the previous experiment in Section 5.4.1 implies. In Figure 5.5(a), gMUA achieves 100% AUR and CMR for  $T_1$  over all range of  $u$ . This is because,  $T_1$  has a step TUF with higher height. Thus, gMUA favors  $T_1$  over others to obtain more utility when it cannot satisfy the critical time of all tasks.

According to Theorem 5.3.4, the system-level AUR must be at least 96%. (For each task  $T_i, \nu_i = 1$ , because all TUFs are step shaped.) We observe that AUR and CMR of gMUA under the condition of Theorem 5.3.4 are above 99.0%. This validates Theorem 5.3.4.

A similar trend is observed in Figure 5.6 for heterogeneous TUFs. We assign step TUFs for  $T_1$  and  $T_4$ , linearly decreasing TUFs for  $T_2$  and  $T_5$ , and parabolic TUFs for  $T_3$  and  $T_6$ . For each task  $T_i, \nu_i$  is set as  $\{1.0, 0.1, 0.1, 1.0, 0.1, 0.1\}$ .

According to Theorem 5.3.4, the system-level AUR must be at least  $0.96 \times (400/25 + 100 \times 0.1/28 + 20 \times 0.1/49 + 100/49 + 30 \times 0.1/41 + 400 \times 0.1/49) / (400/25 + 100/28 + 20/49 +$

$100/49 + 30/41 + 400/49) = 62.5\%$ . In Figure 5.6, we observe that the system-level AUR under gMUA is above 62.5%. This further validates Theorem 5.3.4 for non step-shaped TUFs. We also observe that the system-level AUR and CMR of gMUA degrade gracefully, since gMUA favors as many feasible, high PUD tasks as possible.

## Chapter 6

# Synchronization on Multiprocessors: Lock-Based versus Non-Blocking Methods

We now consider concurrent, serialized object sharing methods for LLREF and gMUA on multiprocessors. The methods include lock-based, wait-free, and lock-free synchronization. Similar to [32], we focus on the common case of concurrent, non-nested, serialized access to simple shared objects. Simple objects (e.g., buffers, queues, stacks) are the predominant shared objects in many embedded applications. For example, Tsigas and Yang observe that almost all synchronization in the SPLASH-2 and the Spark98 kernel benchmark suites are for simple objects [73, 75]. Thus, we focus on simple object sharing under LLREF and gMUA.

For the LLREF algorithm, it is problematic to impose time costs for executing tasks. This is because, LLREF abides by the fairness notion that requires for each task to run at a certain rate in time intervals. (Note that LLREF does not maintain P-fairness [11].) Therefore, any prolonged execution of tasks may violate fairness. In this sense, it is appropriate to consider wait-free synchronization for LLREF, which does not impose any time costs, but incurs space cost. However, the space costs can be minimized using the optimal wait-free synchronization algorithm that we describe in Section 3.1. Thus, we develop wait-free versions of LLREF and gMUA as well, and

derive their space costs using our wait-free algorithm for the SWMR problem.<sup>1</sup>

Despite wait-free synchronization's compatibility with the fairness notion, it is restricted in its use, as described in Section 3.3.1, as opposed to lock-free synchronization. Lock-free objects are known to work well particularly for simple objects like buffers, queues, and lists. In multiprocessor real-time systems, lock-free algorithms have been viewed as impractical, because deducing bounds on retries due to interferences across processors is difficult [43]. Holman *et al.* have bounded retries under Pfair scheduling by exploiting its tight synchrony, and under the observation that lock-free operations take very small time [43]. However, bounding retries is more difficult for asynchronous real-time scheduling, which does not depend on time quanta, e.g., LLREF and gMUA. Thus, we consider an auxiliary method to bound retries as considering the tight synchrony of Pfair scheduled systems, i.e., Pfair scheduling algorithms ensure that no preemption happens within each time quantum, which is highly advantageous for lock-free object sharing.

Inspired by [32], we consider a *non-preemptive area* (or NPA) surrounding each lock-free operation. If there is a scheduling event during the NPA and another task tries to preempt the processor, it will be blocked up to the end of the NPA. Thus, the NPA guarantees finite number of retries for lock-free object sharing. This allows us to use lock-free object sharing even during overloads as opposed to wait-free object sharing. In this Chapter, we present feasible conditions for LLREF and gMUA with NPA-assisted lock-free object sharing.

Although lock-free is efficient for simple objects, lock-based object sharing is still needed to implement more complicated objects. We consider *queue-based spin locks* (or queue locks) for both LLREF and gMUA. In queue locks, a task waits by busy-waiting, or spinning, on “spin variable” (i.e., repeatedly testing its status), and waiting tasks are ordered within a “spin queue”. When a task attempts to acquire a lock, it appends its lock request onto the end of the spin queue. The task at the head of the queue may access the critical section, after which, it updates the spin variable for the next task in the queue so that it stops spinning.

---

<sup>1</sup>Note that our wait-free buffer for the SWMR problem can be used as a building block for developing a wait-free buffer for the MWMR problem.

Similar to [32], we assume that shared object calls are implemented as critical sections accessed via queue locks invoked within non-preemptive regions. It is because if a task is preempted within a critical section, then the waiting times for any tasks that it blocks could significantly increase. Thus, in this Chapter, we consider shared objects implemented with queue-based spin locks within NPA for LLREF and gMUA. As claimed in [32], the idea of the NPA is in line with views expressed by others. The founder of RTLinux recommends accessing simple critical sections non-preemptively in [85], for example. We then derive feasible conditions for LLREF and gMUA under this lock-based scheme.

The rest of the Chapter is organized as follows: Section 6.1 discusses wait-free synchronization under LLREF and gMUA. In Section 6.2, we discuss LLREF and gMUA with NPA-assisted lock-free object sharing. Lock-based synchronization for LLREF and gMUA is discussed in Section 6.3. Finally, we compare the different schemes and identify tradeoffs in Section 6.4.

## 6.1 Wait-Free Buffers

As far as the maximum number of interferences a read operation may suffer from a write operation or/and the maximum number of readers are known, the wait-free buffer algorithm guaranteeing space optimality is usable. Even when either of them is not easy to obtain, the wait-free buffer algorithm is still usable by assuming the unknown value as an infinite, as in Section 3.1.

We consider periodic task arrival and underload cases to compute the minimum required space cost of wait-free buffer for LLREF and gMUA. It is because there exists the feasibility test of LLREF and gMUA only for the periodic task sets. Though, we would like to emphasize again that the wait-free buffer algorithm is available to any case where the maximum number of interferences a read operation may suffer from a write operation or/and the maximum number of readers are known, even with non-periodic task sets.

The maximum number of interferences for each reader's operation is obtained in following Sections and the minimum space cost for wait-free buffer can be obtained by the algorithm for *Wait-*

*Free Buffer size decision Problem* (or WFBP) presented in Section 3.1.

**Figure 6.1:** Reader and Writer Execution Time Line

The maximum number of interferences that a reader may have should be computed. Let  $\rho$  denote the set of wait-free objects shared by tasks. We assume that each wait-free buffer  $w_k \in \rho$  is accessed by at least two tasks and that each task accesses  $w_k$  at most once.  $p_i$  is  $i$ th task's period and deadline for periodic tasks. We assume a writer  $W$  and multiple readers  $R_j$  access  $w_k$ . Since assuming each task accesses  $w_k$  at most once, each reader  $R_j$  and writer  $W$  for  $w_k$  corresponds to a task. Therefore,  $p_{R_j}$  and  $p_W$  imply the period of  $R_j$  and the period of  $W$  respectively.  $N_j^{Max}$  denotes the maximum number of times a writer might interfere with the  $j$ th reader during read operation.

Under underload, the worst case scenario in terms of maximum number of interferences of the reader by the writer task is illustrated in Figure 6.1, as presented in [46]. (We assume that each task's deadline is the same as its period.) This occurs when the first interfering write operation happens as late as possible within the writer's period and the last interfering write operation happens as early as possible within the writer's period. Each  $x$  denotes a write operation. This scenario occurs under LLREF as well as gMUA.

**Corollary 6.1.1** (Maximum number of interferences). *When periodic tasks can be feasibly scheduled by LLREF or gMUA, the upper-bound on the number of interferences  $N_j^{Max}$  that the writer  $W$  might interfere with reader  $R_j$  during read operation is:*

$$N_j^{Max} = \max \left( 2, 1 + \left\lceil \frac{p_{R_j}}{p_W} \right\rceil \right).$$

*Proof.* See [46]

□

When all  $N_j^{Max}$  for each  $w_k$  are obtained, we compute the minimum (optimal) space costs for each  $w_k$  with the algorithm of WFBP presented in [23].

However, the overload caused by relaxed task arrival is difficult to cope with. There are basic requirements for a shared object to be used for overload, i.e., the object operations should be anonymous in the sense that it does not require identities of tasks, as discussed in Section 3.3.1. This is because several jobs of a task may be pending in ready queue under overload and non-anonymous operations are unable to distinguish those jobs since they have the same identifier inherited from the same task. Moreover, it is difficult to assess how many jobs of a task could be pending in the ready queue at the same time.

Unfortunately, our wait-free buffer as well as other wait-free buffers are not anonymous and thus, they have those restriction for the environment allowing overload. It is intuitively understandable that the rationale behind wait-free buffer algorithm is against anonymity. Informally, wait-free buffer algorithm deploys a finite number of internal buffers proportionally to the number of readers and writers, and assigns each internal buffer to each reader and writer to avoid conflict between them, with a certain clever way to handle data replication across those internal buffers. In other words, it starts from assumption that the fixed number of readers and writers is known in advance.

## 6.2 Lock-Free Objects

Lock-free objects are a good candidate of non-blocking synchronization under overload because most well-known lock-free objects are anonymous. However, it is additionally required to assure that the lock-free retry should be bounded under overload as well as underload.

The boundness of retry is derived under several assumptions, which are made from practical observation of lock-free objects and potential interferences associated with applied scheduling algorithms as well as task arrival patterns. In [43], the observation is introduced that lock-free operations typically are very short in comparison to the length of a time quantum, that their scheduling algorithm, Pfair, is based upon. The following assumptions are the variants of theirs to bound retry

for using lock-free objects with our scheduling algorithms.

- (1) **Interference Assumption (IA)** Any pair of concurrent accesses to the same object may potentially interfere with each other.
- (2) **Retry Assumption (RA)** A retry can be caused only by the completion of some object access. This bounds the number of retries to at most the number of concurrent accesses to an object and prevents two tasks from livelocking due to repeated mutual interferences.
- (3) **Preemption Assumption (PA)** A single object access will be preempted for a finite number of times.

These assumptions help to establish bounds on retries in multiprocessor real-time systems even with interferences across processors. Holman *et al.* have bounded retry in Pfair scheduling by exploiting its tight synchrony and observation that lock-free operations take very small time [43]. However, bounding retry is more difficult for asynchronous real-time scheduling not depending on time quanta, e.g., gMUA, LLREF, etc.

One example that the boundness of retry collapses is in Figure 6.2. It shows task  $T_i$  running on one processor in a multiprocessor system. Even if assuming (RA) holds, other tasks' scheduling events causes preemptions and hence, retry may never end. In other words, a condition for the assumption (PA) is required to hold with asynchronous real-time scheduling algorithms for use of lock-free objects.

**Figure 6.2:** An Example of Continuous Retry

To prevent from the continuous retry, we suggest the NPA surrounding each lock-free operation.

The objective of NPA is to make (PA) hold by ensuring that the access to a single object is never preempted. If there is a scheduling event during the NPA and another task tries to preempt the processor, it will be blocked up to the end of NPA.

### 6.2.1 Non-Preemptive Area

We follow some notations as used in [43]. Let  $\rho$  denote the set of objects shared by tasks. We assume that each object  $r_k \in \rho$  is accessed by at least two tasks and that each task accesses  $r_k$  at most once. We assume that each access to  $r_k$  has a base cost of  $b(r_k)$  and a retry cost of  $s(r_k)$ . We are assuming that the costs of all operations (e.g., read, write) of  $r_k$  are the same as in [43], which could be eliminated at the price of slightly more complicated notion and analysis. The number of tasks accessing  $r_k$  is denoted by  $A(r_k)$ . Note that  $A(r_k)$  never exceeds the number of tasks,  $N$ , because we assume each task accesses  $r_k$  at most once.  $p_i$  is  $i$ th task's period for periodic tasks or minimum time interval between adjacent  $i$ th task's releases for sporadic tasks.  $e_i$  is the  $i$ th task's base execution cost, i.e., the worst-case cost of a single job of  $T_i$  without considering shared-object accesses. We assume a job of  $T_i$  accesses at most  $R_i$  number of shared objects. The set of shared objects that  $T_i$  accesses is denoted by  $\gamma_i$ . We assume  $M$  identical processors in the system.

We attempt to make three assumptions aforementioned including (IA), (RA), and (PA), with the NPA. Under all assumptions, the time cost of NPA may be extended by retry.

**Theorem 6.2.1** (Maximum Execution Cost of NPA). *When  $T_i$  accesses a lock-free object  $r_k$  with NPA-assisted sharing mechanisms,  $L_{i,k}^{LF}$ , the maximum execution time of the NPA is:*

$$L_{i,k}^{LF} = s(r_k) \cdot \{\min(A(r_k), M) - 1\} + b(r_k),$$

where  $r_k \in \gamma_i$ .

*Proof.* NPA can be extended by continuous retry, which occur from interferences. There are  $A(r_k)$  number of tasks which access  $r_k$  and thus, may cause interferences. Therefore, the maximum possible interferences for accessing  $r_k$  is  $A(r_k)$  and it cannot exceed the number of processors  $M$ .

Based on it, the time of possible retry is at most  $s(r_k) \cdot \{\min(A(r_k), M) - 1\}$ . In addition, the base access cost,  $b(r_k)$ , is also considered. Preemption is prohibited by the NPA surrounding each lock-free operation.  $\square$

Note that the NPA causes blocking and thus the execution cost of the NPA affects feasibility analysis considering blocking times.

To the best of our knowledge, the feasibility analysis of more relaxed task arrival patterns than periodic one for multiprocessor real-time scheduling does not exist due to difficulty. More specifically, most of feasibility analysis for two well-known scheduling algorithms including Pfair and global EDF have been on periodic tasks. Even though NPA-assisted lock-free approach allows sporadic tasks which may cause overloads where lock-free retry can be bounded, we also focus on the feasibility analysis of periodic tasks in following sections.

## 6.2.2 Lock-Free Objects For gMUA

gMUA is designed to ensure the feasibility of the global EDF so that gMUA's and global EDF's feasibilities for periodic tasks are essentially similar. There have been three research efforts to obtain much tighter utilization bound within which the global EDF guarantees to meet all deadlines and it is known that each of them does not dominate the others in [13].

NPA-assisted lock-free sharing mechanisms causes blocking unlike the pure lock-free sharing mechanisms. As described in [32], if  $b_i$  denotes the maximum execution time cost of any NPA of  $T_i$ , and tasks are ordered in the non-decreasing order of their relative deadlines, then each job of  $T_i$  can be blocked for at most  $B_i = \max_{i+1 \leq j \leq n} \{b_j\}$ .  $b_i$  can be obtained by  $\max_{\forall r_k \in \gamma_i} \{L_{i,k}^{LF}\}$ . Under gMUA just as preemptive EDF, guaranteeing that each job of each task  $T_i$  can be allocated  $e_i$  units of time in the interval  $[a + B_i, a + p_i]$  is sufficient to ensure the feasibility, where  $a$  is the arrival time of the job and  $B_i$  is as defined above. Thus, if  $p_i - B_i \geq e_i$ , for all  $T_i$ , and all  $T_i$  with the deadline  $p_i - B_i$  instead of  $p_i$  pass the feasibility test, given in [36], [10], or [13], then all  $T_i$  are feasible.

NPA-assisted lock-free object sharing is an approach that is conceptually located between lock-based and lock-free mechanisms. It takes advantage of mutual exclusion by NPA to ensure the boundness of lock-free retry for real-time systems. It is analogous to the approach that lock-based sharing takes to ensure the boundness of spinning, which will be described in following sections. As expected, NPA introduces a side effect as lock-based synchronization, e.g., blocking. Though, it is still interesting in that the blocking time is bounded. More details are discussed in Section 6.4

### 6.2.3 Lock-Free Objects For LLREF

LLREF is an optimal real-time scheduling algorithm satisfying all time requirements if the total utilization demand of tasks is less than the capacity of processors. We consider NPA-assisted lock-free sharing for LLREF.

As compared with gMUA as well as EDF, NPA under LLREF causes more times of blocking. It is because the priority order of jobs under LLREF changes over their execution as opposed to the case where tasks are scheduable under gMUA and EDF. When tasks are scheduable under gMUA and EDF, the priority order of jobs never changes so that a preempted job  $A$  cannot preempt a job  $B$  that has preempted the job  $A$  before. However, it happens under LLREF that allows more scheduling events, e.g., event  $C$  and  $B$  as introduced in Chapter 4. Whenever preemption occurs, blocking may arise when the preempted task is within the NPA and no processor is available. Therefore, each job can be blocked for at most  $B = \max_{\forall i \forall k} \{L_{i,k}^{LF}\}$ .

**Theorem 6.2.2** (Blocking under LLREF with NPA-assisted mechanisms). *When a set of periodic tasks  $T_1, \dots, T_N$  has deadline equal to period, the maximum times that  $T_i$  can be blocked within its period  $p_i$  is:*

$$n_i^B = \left\lceil \frac{N+1}{2} \right\rceil \cdot \left( 1 + \sum_{j=1}^N \left\lceil \frac{p_i}{p_j} \right\rceil \right).$$

*Proof.* Each T-L plane is constructed between two consecutive events of task release. The number of task releases within period  $p_i$  is  $\sum_{j=1}^N \left\lceil \frac{p_i}{p_j} \right\rceil$ . Thus, there can be at most  $1 + \sum_{j=1}^N \left\lceil \frac{p_i}{p_j} \right\rceil$  number of T-L planes within  $p_i$ . It is known that at most  $N + 1$  events can occur in each T-L plane in

Theorem 4.2.12. Since blocking of  $T_i$  happens only when  $T_i$  attempts to preempt,  $T_i$  can be blocked at most  $\lceil \frac{N+1}{2} \rceil$  in each T-L plane. Therefore, within the period  $p_i$ ,  $T_i$  can be blocked at most  $\lceil \frac{N+1}{2} \rceil \cdot (1 + \sum_{j=1}^N \lceil \frac{p_i}{p_j} \rceil)$  times.  $\square$

**Theorem 6.2.3** (LLREF feasibility with NPA-assisted lock-free synchronization). *A set of periodic tasks  $T_1, \dots, T_N$ , all with deadline equal to period, is guaranteed to be feasible on  $M$  processors using LLREF if*

$$\sum_{i=1}^N \frac{Bn_i^B + \sum_{\forall r_k \in \gamma_i} L_{i,k}^{LF} + e_i}{T_i} \leq M,$$

where  $L_{i,k}^{LF} = s(r_k)(\min\{A(r_k), M\} - 1) + b(r_k)$ .

*Proof.* When NPA-assisted lock-free objects are shared, each task's execution time extended by lock-free retry and blocking time caused by the NPA should be considered.  $T_i$ 's blocking happens at most  $n_i^B$  times and each blocking time costs at most  $B$  as described in Theorem 6.2.1 and 6.2.2. Besides, when  $T_i$  accesses  $r_k$  within the NPA, it can be extended at most  $L_{i,k}^{LF}$  as in Theorem 6.2.1. Therefore, the execution time is extended by  $Bn_i^B + \sum_{\forall r_k \in \gamma_i} L_{i,k}^{LF}$  in maximum. When the total utilization demand obtained with tasks' extended execution times is less than the capacity of processors, tasks meet deadlines under LLREF.  $\square$

Note that LLREF does not consider the case that the total utilization demand exceeds the capacity of processors. Thus, using shared objects, including lock-free shared objects, with LLREF are under the assumption that the total utilization demand never exceeds the capacity of processors.

### 6.3 Lock-Based Synchronization

We consider that shared objects implemented with queue-based spin locks within the NPA as in [32]. NPA-assisted queue lock causes blocking as aforementioned.

### 6.3.1 Non-Preemptive Area

As described in [32], under NPA-assisted, queue-lock based synchronization, a job may be blocked under two scenarios: (1) the job is executing and requires access to a resource for which one or more jobs have already enqueued their requests onto the spin queue, or (2) the job becomes ready when one or more lower-priority jobs are in their NPA and no processor is available.

**Theorem 6.3.1** (Maximum Execution Cost of NPA). *When  $T_i$  accesses a queue-based spin lock object  $r_k$  with NPA-assisted sharing mechanisms,  $L_{i,k}^{QS}$ , the maximum execution time of the NPA is:*

$$L_{i,k}^{QS} = q(r_k) \cdot \min(A(r_k), M),$$

where  $q(r_k)$  is the access cost to  $r_k$ ,  $M$  is the number of processors in system, and  $r_k \in \gamma_i$ .

*Proof.* Since  $A(r_k)$  tasks accessing  $r_k$  may cause interferences, the maximum possible interferences is  $A(r_k)$  and it cannot exceed the number of processors  $M$ . Based on it, the maximum execution cost of completing access to  $r_k$  is  $q(r_k) \cdot \min(A(r_k), M)$ . Preemption is prohibited by the NPA surrounding each lock-free operation.  $\square$

For the same reason we presented in Section 6.2.3, we consider feasibility analysis on periodic task sets with lock-based synchronization for gMUA and LLREF.

### 6.3.2 Lock-Based Synchronization for gMUA

Since gMUA ensures the feasibility of the global EDF, their feasibility tests for periodic tasks are identical. As presented in Section 6.2.2, each job of  $T_i$  can be blocked for at most  $B_i = \max_{i+1 \leq j \leq n} \{b_j\}$ , when the maximum execution cost of any NPA of  $T_i$  is  $b_i$  and tasks are ordered in the non-decreasing order of their relative deadlines. Each  $b_i$  can be gained by  $\max_{r_k \in \gamma_i} \{L_{i,k}^{QS}\}$ . If each job of each task  $T_i$  can be allocated  $e_i$  units of time in the interval  $[a + B_i, a + p_i]$ , where  $a$  is the arrival time of the job, each task is feasible. Thus, if  $p_i - B_i \geq e_i$  for all  $T_i$ , and all  $T_i$  with

the deadline  $p_i - B_i$  instead of  $p_i$  pass the feasibility test given in [36], [10], or [13], then all  $T_i$  are feasible.

The feasibility test with NPA-assisted lock-based synchronization is analogous to that with NPA-assisted lock-free except that each  $B_i$  is computed based on different object access times. The analogy of two approaches comes from the fact that both are assisted with the NPA.

### 6.3.3 Lock-Based Synchronization for LLREF

We consider NPA-assisted lock-free shared objects under LLREF. More frequent scheduling events of LLREF as compared to gMUA and EDF cause more blocking and thus, it affects the feasible condition. Whenever preemption occurs, blocking follows when the preempted task is within the NPA and no processor is available.

**Theorem 6.3.2** (LLREF feasibility with NPA-assisted lock-based synchronization). *A set of periodic tasks  $T_1, \dots, T_N$ , all with deadline equal to period, is guaranteed to be feasible on  $M$  processors using LLREF if*

$$\sum_{i=1}^N \frac{Bn_i^B + \sum_{\forall r_k \in \gamma_i} L_{i,k}^{QS} + e_i}{T_i} \leq M,$$

where  $L_{i,k}^{QS} = q(r_k) \cdot \min(A(r_k), M)$ .

*Proof.* When NPA-assisted queue-lock objects are shared, each task's execution time extended by the spin queue and blocking time caused by the NPA should be considered.  $T_i$ 's blocking happens at most  $n_i^B$  times as in Theorem 6.2.2 and each blocking time costs at most  $B = \max_{\forall i \forall k} L_{i,k}^{QS}$ . Besides, when  $T_i$  accesses  $r_k$  within the NPA, it can be extended at most  $L_{i,k}^{QS}$  as in Theorem 6.3.1. Therefore, the execution time is extended by  $Bn_i^B + \sum_{\forall r_k \in \gamma_i} L_{i,k}^{QS}$  in maximum. When the total utilization demand obtained with tasks' extended execution times is less than the capacity of processors, tasks meet deadlines under LLREF.  $\square$

## 6.4 Tradeoffs

The tradeoff between the presented wait-free method and lock-free/lock-based methods is one of space and time costs. Wait-free object sharing costs space, but incurs no additional (blocking or retry) time costs. Further, it allows the full capacity of all processors to be utilized. However, it is restricted to the case of bounded number of jobs, in contrast with lock-free and lock-based which allows unbounded number of jobs.

We now discuss the tradeoff between the lock-free and lock-based methods that we present. Lock-free object sharing does not use lock mechanisms which can potentially cause blocking, unlike lock-based object sharing. However, blocking may happen with the lock-free objects as well as the lock-based ones that we present, since both are designed to be assisted with the NPA. Despite the blocking caused by the NPA that is common to both, it is still worth discussing the tradeoffs between NPA-assisted lock-free objects and NPA-assisted queue lock objects.

We focus on the implementation of objects within the NPA (but not on the implementation of the NPA itself). First, lock-free sharing is an optimistic approach as opposed to queue-based spin locks that must lock resources even when no interference occurs. The optimism prohibits the time costs for unnecessary operations. Second, lock-free sharing mechanisms are implemented at the application level, whereas the implementation of the NPA is through kernel calls. The implementation at the application level is likely to help reduce the overhead of operations since it does not invoke kernel, which is advantageous with respect to timeliness.

For these reasons, lock-free object sharing is likely to be superior to queue-lock object sharing when simple objects (which are implementable in lock-free mechanisms) are considered. On the other hand, queue-lock object sharing is likely to be superior when more complicated objects (e.g., map, etc.) are considered.

### 6.4.1 Numerical Analysis under LLREF

Specifically, Theorem 6.2.3 and 6.3.2 show feasible conditions under LLREF with lock-free and lock-based synchronization, respectively. We observe that the forms of two feasibility tests are identical when the access times to shared objects are assumed similar, i.e.,  $s(r_k) = b(r_k) = q(r_k), \forall k$ . It is because both sharing mechanisms are assisted by the NPA to ensure the boundedness of access time by preventing preemption temporarily. Therefore, the performance difference between NPA-assisted lock-free objects and NPA-assisted queue-lock objects is primarily determined from the difference between their object access times. (It is not a claim generalized for every case, since NPA-assisted mechanisms considered here should not be the only approach to implement lock-free and lock-based sharing.)

In [32], they set contention-free costs for object operations in  $1.3 - 6.5 \mu s$  range for both lock-free and queue-lock approaches by their measurement. We vary the costs,  $s(r_k)$ ,  $b(r_k)$ , and  $q(r_k)$ , from 1.0 to 6.0 to see the effect. The performance metric is the total utilization inflation denoting the difference between total utilization without shared objects and that with shared objects, which could be interpreted as synchronization overhead or utilization loss as described in [32].

We assume four processors and 20 tasks. Each task has three objects shared with others and each objects are shared by 10 tasks. Task's execution time  $e_i$  is uniformly distributed in different ranges,  $[0.5, 5]$ ,  $[1, 10]$ ,  $[1.5, 15]$ , or  $[2, 20]$  *ms*. As Theorem 6.2.3 and 6.3.2 imply, task's execution time cost is a primary element that affects the utilization inflation. Task's period is set by  $e_i/u_i$ , where  $u_i$  is task's utilization demand.  $u_i$  is uniformly distributed in the range  $(0.0, 0.2]$  and thus, the total utilization demand without shared objects never exceeds the capacity of processors. Each data was gained with 5,000 samples.

Figure 6.3 shows that the higher shared-object access time cost and the lower task's execution time cost cause higher increase in total utilization. It suggests that the shared-object access time cost is an important element to select between lock-free and queue-lock shared objects under LLREF. In addition, it also implies that the selected shared object becomes more appropriate as tasks have

**Figure 6.3:** LLREF-scheduled Synchronization Overhead for NPA-assisted Mechanism with Varying Tasks' Execution Time Costs

more execution time costs since it makes the synchronization overhead negligible.

**Figure 6.4:** LLREF-scheduled Synchronization Overhead for NPA-assisted Mechanism with Varying Number of Shared Objects

Figure 6.3 illustrates the increase in total utilization with varying shared-object access time cost and number of objects. As expected, the more objects are shared, the higher synchronization overhead does rise. It is also observed in Figure 6.3 that there is a number, more shared objects than which does not have any impact on performance as Theorem 6.2.3 and 6.3.2 indicate.

Note that the comparison results in this section show the worst case of both sharing mechanisms. Thus, the actual performance comparison by real implementation could be different.

## 6.4.2 Numerical Analysis under gMUA

We believe the performance of NPA-assisted synchronization mechanisms under gMUA is similar to that of queue-based spin lock mechanisms under EDF evaluated in [32], since gMUA behaves as EDF within feasible utilization bound. In this section, we evaluate NPA-assisted lock-free and queue lock sharing under gMUA.

Section 6.2.2 and 6.3.2 show that the forms of feasible conditions of lock-free and lock-based synchronization for gMUA are identical when the access times to shared objects are assumed similar, i.e.,  $s(r_k) = b(r_k) = q(r_k), \forall k$ . As previously discussed, it is because both sharing mechanisms are assisted by the NPA to ensure the boundness of access time by preventing preemption temporarily.

We set contention-free costs for object operations from 1.0 to 6.0 to see its effect. Task's execution time  $e_i$  is uniformly distributed in different ranges, [50, 500], [100, 1000], [150, 1500], or [200, 2000]  $\mu s$ . Task's period is set by  $e_i/u_i$ , where  $u_i$  is task's utilization demand.  $u_i$  is uniformly distributed in the range (0.0, 0.2] and thus, the total utilization demand without shared objects never exceeds the capacity of processors. Each data was gained with 5,000 samples.

**Figure 6.5:** gMUA-scheduled Synchronization Overhead for NPA-assisted Mechanism with Varying Tasks' Execution Time Costs

Figure 6.5 and 6.6 shows the similar trend as we observed in cases of LLREF. The higher shared-object access time cost, the lower task's execution time cost, and more number of shared objects

**Figure 6.6:** gMUA-scheduled Synchronization Overhead for NPA-assisted Mechanism with Varying Number of Shared Objects

are factors to increase the synchronization overhead. However, the inflation in total utilization under gMUA is less than that under LLREF.

gMUA's less frequent scheduling events and blocking are advantageous to its utilization inflation compared to LLREF. However, since its feasible utilization bound is less than that of LLREF, small amount of utilization inflation could affect its feasibility.

# Chapter 7

## Past and Related Works

Real-time scheduling and synchronization — the dissertation’s problem space — are fundamental problems in real-time computing. Recent technological advances in multiprocessor architectures and the increasing demand of real-time applications for cost-effective computing capacity have resulted in intense research efforts on single-processor real-time scheduling, synchronization, and their extension to multiprocessor systems. In this Chapter, we survey past efforts that are related to the dissertation’s research, focusing on TUF real-time scheduling, non-blocking synchronization, and multiprocessor real-time scheduling, and contrast them with our work. Note that some related past works were already discussed in previous Chapters as appropriate.

### 7.1 Time/Utility Function Real-Time Scheduling

UA scheduling was first introduced by Jensen [37] in a classified military system. The first public reference was written by Jensen, Locke, and Tokuda in [47] and was later elaborated on in Locke’s thesis [56] and Clark’s thesis [27]. Subsequently, there have been very few efforts that built upon Locke’s and Clark’s algorithms. Most of the subsequent efforts focussed on downward step TUFs and sought to improve the timeliness behavior during overloads [49, 60]. Exceptions include [21,

79], which consider non-step TUFs.

While Locke's algorithm allowed almost arbitrary TUF shapes, it was restricted to resource-independent activities. Clark's algorithm, called DASA, advanced Locke's model by allowing activities to mutually exclusively share non-CPU resources under the single-unit resource request model. DASA also provides assurances on timeliness behavior such as optimal timeliness during under-load situations. However, DASA is restricted to downward step TUFs. Algorithms in [21, 79] allow non-step, but unimodal TUFs. Further, they do not allow sharing of non-CPU resources.

In spite of the generality and superiority of TUFs and UA scheduling, they have drawbacks that have apparently been impediments to their widespread usage. The most significant drawbacks include: (1) lack of more general timeliness assurances of TUF/UA systems, beyond the special-case assurance of optimal timeliness during under-loads for step TUFs; (2) lack of tool support for composing TUFs and for conducting UA timing analysis; (3) lack of UA algorithms that consider quality of service dimensions of embedded systems other than timeliness, such as power consumption and memory management; and (4) UA algorithms' higher overhead than priority/deadline-based algorithms.

Recent research on UA scheduling has largely overcome these drawbacks. Several new UA scheduling algorithms have been recently developed. Examples include Li's assurance-driven UA scheduling algorithms and protocols [50, 53], Wu's energy-efficient UA scheduling algorithms [81, 82, 84], etc.

Li's UA scheduling algorithms and protocols break significant new ground by providing more general assurances on timeliness behavior of TUF/UA systems such as assurances on individual activity timeliness behavior and system-wide, collective timeliness behavior. Li's algorithms and protocols consider stochastic activity models, where activity execution times and inter-arrival times are stochastically described. In particular, the algorithms and protocols consider a stochastic, activity arrival pattern called PUAM, which is a probabilistic generalization of the unimodal arrival model, and which subsumes most traditional arrival models (e.g., frame-based, periodic, sporadic,

unimodal) as special cases. The algorithms and protocols allow activities to be subject to unimodal TUFs, and to mutually exclusively share non-CPU resources under the single-unit request model.

Wu's algorithms extend the TUF/UA paradigm in a quality of service dimension that is critical for emerging, mobile and portable, battery-powered embedded systems: *energy consumption*. Wu's algorithms solve, for the first time, the overlapped problem space that intersects: (1) UA scheduling under TUF time constraints, providing general assurances on individual and collective timeliness behavior; (2) scheduling activities under mutual exclusion resource constraints; and (3) CPU scheduling for reduced and bounded system-level energy consumption.

Wu presents a class of polynomial-time, DVS (dynamic voltage scaling)-based, UA scheduling algorithms toward this objective. The algorithms are called EUA [82], ReUA [84], and EUA $\star$  [81]. EUA provides stochastic assurances on timeliness behavior and maximizes activities' attained utility and system-level energy efficiency. ReUA extends EUA's step TUF model with non-increasing TUFs, and extends EUA's independent task model with mutually exclusive resource sharing. Further, EUA $\star$  extends EUA's and ReUA's periodic arrival model with unimodal arrival.

Wu's RUA algorithm [80] makes a major contribution by extending TUF/UA scheduling to mutually exclusive resource sharing under the *multi-unit* resource request model. The multi-unit model generalizes the single-unit model and subsumes the reader/writer lock model and the AND, OR, and AND-OR request models as special cases. Similar to DASA, RUA achieves optimal timeliness during under-loads, and upper bounds the resource access blocking time.

More detailed survey is found in [64].

## 7.2 Non-Blocking Synchronization

Wait-free buffer algorithms to solve single-writer and multi-reader problem have been intensively studied. In [48], Kopetz and Reisinger presented a wait-free buffer algorithm based on circular buffers, where buffer sizes in proportional to worst-case preemptions are used. In [19], Chen and

Burns presented a wait-free buffer, where the space complexity is  $O(N + 2)$  when the number of readers is known as  $N$ . In [46], Huang *et al.* improve the time and space costs of Chen's protocol based on temporal characteristics of the systems. Our wait-free buffer considers the same task model as Huang's, and we propose the first analytical approaches to compute the minimum and optimal space cost [23]. Besides, whereas most past efforts on non-blocking real-time synchronization have focused on deadline time constraint and deadline-based timeliness optimality criteria, we showed the advantages of our wait-free approaches for TUF/UA scheduling [25].

Wait-free buffer algorithm for multi-writer and multi-reader problem is also challenging to researchers. Theoretically, with  $n$  readers and  $m$  writers, the minimum number of needed buffers is  $n + m + 1$  for the MWMR problem [66]. However, most wait-free solutions for the MWMR problem need  $n \times m$  number of atomic buffers, where each atomic buffer has to solve the single-writer/single-reader (SWSR) problem [38, 77]. Anderson and Holman presented wait-free buffers based on specific system models, such as quantum-based and priority-based, although it requires the reduced buffer size for MWMR problem [7]. Their space complexity depends not on the number of readers but on the number of processors. In [70], Sundell and Tsigas tried to present a wait-free buffer for the MWMR problem by using tasks' time information that is available in real-time systems. They have developed an algorithm to bound the increment of time-stamps to mark the latest updated data. However, their algorithm is limited to the system where writers have the highest priority on the host processor and no two writers execute on one host processor.

The wait-free objects are often implemented by adopting helping mechanisms instead of replication of data. Anderson proposed several wait-free object-sharing schemes for real-time uniprocessors and multiprocessors based on helping schemes. Their basic idea is that the task which announces its intention to share should help the other tasks to complete execution. Their uniprocessor implementation are pretty similar to the priority inheritance protocol (PIP) and the priority ceiling protocol (PCP) [5].

More complicated wait-free objects than buffers have been studied. Tsigas and Zhang presented an efficient algorithmic implementation of wait-free queues. Their algorithm shows  $O(M + N)$  space

complexity where  $N$  is the maximum number of concurrent tasks that the queue supports and  $M$  is the size of the queue [74]. Their group also presented a simple wait-free snapshot algorithm by making use of timing information that is available and necessary to the scheduler that schedules the tasks of real-time systems [71]. Anderson also suggested snapshot, which he called composite registers in [2].

There have been researches to apply lock-free object sharing mechanism to real-time systems. In [6], Anderson *et al.* showed how to bound the retry loops of lock-free protocols through judicious scheduling. These requirements are not only for wait-free buffer, but also for all non-blocking sharing including lock-free sharing. Their group presented implementation of lock-free objects in single processors and multiprocessors in which tasks are executed using a scheduling quantum [4]. Their approaches are based on the assumption that each task can be preempted at most once across two object call, so that each object call can be retired at most once.

Several lock-free objects exist for the MWMR problem [58], which are anonymous and space-efficient. Michael and Scott presented the most famous lock-free queue in [58]. Valois presented lock-free linked list supporting insertions and deletions anywhere in a list by using CAS primitives [76]. Treiber presented lock-free stack algorithms [72]. We presented that lock-free objects are also effective for UA scheduling on single processors in [24].

Practical consideration to implement non-block objects has been gaining attention. Hohmich and Hartig presented a pragmatic methodology for designing nonblocking real-time systems, which are simple but reasonable. They use a combination of lock-free and wait-free synchronization techniques and stated which technique should be applied in which situation [41]. Weaker non-blocking objects than wait-free and lock-free ones have been suggested in [14, 39]. A synchronization technique is *obstruction-free* if it guarantees progress for any thread that eventually executes in isolation. This property is weaker than lock-free because it does not guarantee progress when two or more conflicting threads are executing concurrently. On the other hand, another weaker class of *almost non-blocking* data structures was proposed, which block only if more than some number  $N$  of threads attempt to simultaneously access the same data structure. Those weaker non-blocking

objects are often sufficient to requirement.

### 7.3 Multiprocessor Real-Time Scheduling

In recent years, real-time scheduling has been studied considerably. Carpentar *et al.* [15] have catalogued multiprocessor real-time scheduling algorithms considering the degree of job migration. The Pfair class of algorithms [11] that allow full migration have been shown to achieve a feasible utilization bound (below which all tasks meet their deadlines) that equals the total capacity of all processors — thus, they are theoretically optimal. However, Pfair algorithms incur significant overhead due to their quantum-based scheduling approach [31]. Thus, we present an optimal real-time scheduling algorithm, LLREF, that does not depend on time quanta unlike Pfair in [22].

On the other hand, scheduling algorithms other than Pfair (e.g., global EDF) have also been studied though their feasible utilization bounds are lower. Global EDF scheduling on multiprocessors is subject to the “Dhall effect” [33], where a task set with total utilization arbitrarily close to 1.0 cannot be scheduled satisfying all deadlines. To overcome this, researchers have studied global EDF’s behavior under restricted individual task utilizations. For example, on  $M$  processors, Srinivasan and Baruah show that when the maximum individual task utilization,  $u_{max}$ , is bounded by  $M/(2M - 1)$ , EDF’s utilization bound is  $M^2/(2M - 1)$  [68]. Lin presented a variant of EDF, *Earliest Deadline until Zero Laxity* (or EDZL), which allows another scheduling event to split tasks [55]. In [36], Goossens *et al.* show that EDF’s utilization bound is  $M - (M - 1)u_{max}$ . This work was later extended by Baker for the more general case of deadlines less than or equal to periods in [10]. In [13], Bertogna *et al.* show that Baker’s utilization bound does not dominate the bound of Goossens *et al.*, and vice versa.

In fact, EDF’s non-optimality for multiprocessors was expected. Dertouzos *et al.* proved that if we do not have a-priori knowledge of any one of the following parameters: (1) deadlines, (2) execution times, or (3) the releasing times for tasks, then there exists no optimal algorithm for multiprocessor like EDF for single-processor [30]. The EDF scheduling decision only requires the awareness of

deadlines.

While most of these past works focus on the hard real-time objective of always meeting all deadlines, recently, there has been efforts that consider the objective of bounding the tardiness of tasks. In [67], Srinivasan and Anderson derive a tardiness bound for a suboptimal Pfair scheduling algorithm. In [3], for a restricted migration model, where migration is allowed only at job boundaries, Andersen *et. al* present an EDF-based partitioning scheme and scheduling algorithm that ensures bounded tardiness. In [31], Devi and Anderson derive the tardiness bounds for global EDF when the total utilization of a task system may equal the number of available processors. We present the first UA scheduling algorithm, gMUA, for multiprocessors in [26].

There have been several efforts on synchronization for multiprocessor real-time systems. Rajkumar presented an extension of priority-ceiling protocol for partitioned rate-monotonic scheduling in [63]. Wang proposed priority inheritance spin locks for multiprocessor real-time systems in [78]. The synchronization under global EDF has first been considered recently in [32]. They consider lock-free as well as lock-based schemes for soft and hard real-time systems based on global EDF. Holman *et al* considered lock-free and lock-based shared objects for Pfair-scheduled multiprocessor systems in [43] and [42] respectively. They utilized the tight synchrony of Pfair scheduled system for object sharing.

# Chapter 8

## Conclusions and Future Work

This dissertation presents UA real-time scheduling and synchronization algorithms for single and multiprocessors. The dissertation focuses on dynamic real-time systems that operate in environments with run-time uncertainties including those on activity execution times and arrival behaviors. We consider the TUF timing model for specifying application time constraints, and the UA optimality criteria of satisfying lower bounds on accrued activity utility, and maximizing the total accrued utility.

The dissertation presents a class of novel algorithmic solutions in this problem space, including a space-optimal wait-free synchronization algorithm, an optimal real-time scheduling algorithm for multiprocessors, and a UA scheduling algorithm for multiprocessors. Further, the dissertation studies various synchronization methods for UA scheduling on single and multiprocessors, including lock-based, lock-free, and wait-free synchronization, develops feasible conditions, and identifies the concomitant tradeoffs.

We now summarize the dissertation's contributions and identify future work.

## 8.1 Summary of Contributions

*UA Scheduling and Synchronization for Single Processors.* We consider the single-writer/multiple-reader problem that occurs in many embedded real-time systems. We present an analytical solution for the problem of determining the absolute minimum buffer requirement of wait-free protocols — the first such buffer lower bound for wait-free protocols which is independent of any scheduling algorithm. We also show that the buffer costs required by previous best algorithms for this problem, including Chen’s and NBW protocols can also be obtained by our solution, which subsumes them as special cases. Further, we present a wait-free protocol that uses the minimum buffer size that is determined by our analytical solution. Our numerical evaluation studies and measurements of the protocol implementation in an RTOS kernel validate our theoretical results.

We then introduce for the first time, wait-free synchronization for UA real-time scheduling. We develop wait-free versions of two single-processor lock-based UA algorithms, RUA and DASA, using our space-optimal wait-free protocol (for the single-writer/multiple-reader problem). We establish upper bounds on the maximum possible increase in activity utility that is possible with wait-free, compared to their lock-based counterparts. Our implementation measurements on a POSIX RTOS show that during under-loads, wait-free algorithms yield optimal utility for step TUFs and significantly higher utility (than lock-based) for non-step TUFs.

For single processor real-time systems, we finally consider non-blocking synchronization for real-time applications that are subject to resource overloads and arbitrary activity arrivals. We consider lock-free synchronization for the multi-writer/multi-reader problem that occurs in such systems (due to unbounded number of jobs and otherwise), for which wait-free synchronization is not possible. We establish the tradeoffs between lock-free and lock-based object sharing under the unimodal arbitrary arrival model, including the conditions under which activity timeliness utility is greater under lock-free than under lock-based, and the upper bound on this utility increase — the first such result. Our implementation measurements on a POSIX RTOS strongly validates our theoretical results.

***UA Scheduling and Synchronization for Multiprocessors.*** We present an optimal multiprocessor real-time scheduling algorithm that satisfies all activity deadlines during under-loads (a special case of UA criteria), which is not based on time quanta. The algorithm called LLREF, is designed based on our novel technique of using the T-L plane abstraction for reasoning about multiprocessor scheduling. We show that scheduling for multiprocessors can be viewed as repeatedly occurring T-L planes, and correct scheduling on a single T-L plane leads to the optimal solution for all times. We analytically establish the optimality of LLREF. We also establish that the algorithm overhead is bounded in terms of the number of scheduler invocations, which is validated by our experimental (simulation) results.

We also present a global UA multiprocessor scheduling algorithm, called gMUA. The algorithm considers activities that are subject to TUF time constraints, stochastically-expressed variable execution time demands, and resource overloads. We establish that gMUA achieves optimal total utility for the special case of step TUFs and when the total utilization demand does not exceed global EDF's feasible utilization bound, probabilistically satisfies activity utility lower bounds, and lower bounds the system-wide total accrued utility. We also show that the algorithm's utility lower bound satisfactions have bounded sensitivity to variations in execution time demand estimates, and that it is robust against a variant of the Dhall effect. When activity utility lower bounds cannot be satisfied (due to increased utilization demand), gMUA maximizes total utility, while gracefully degrading timeliness. Our simulation experiments validate our theoretical results and confirm the algorithm's effectiveness. Our method of transforming task stochastic demand of activities into actual execution time allocation is independent of gMUA and can be applied in other algorithmic contexts, where similar (stochastic scheduling) problem arises.

Finally, we consider lock-based, lock-free, and wait-free synchronization methods for both LLREF and gMUA. We first show the wait-free synchronization (which is appropriate for only bounded number of jobs) does not incur significant time costs, but only space costs, which helps to maintain the fairness notion of LLREF. Further, we establish the minimum (optimal) required space costs for LLREF and gMUA with our space-optimal wait-free synchronization algorithm. In contrast to wait-free, lock-based and lock-free synchronization allow unbounded number of jobs and

overloads. We introduce non-preemptive area to bound the time cost of lock-based and lock-free synchronization under LLREF and gMUA, and derive feasible conditions for satisfying utility lower bounds. Further, we observe the tradeoff between lock-based and lock-free synchronization for LLREF and gMUA, i.e., lock-free object sharing is likely to be superior to lock-based sharing when simple objects are considered, whereas lock-based object sharing is likely to be superior when more complicated objects are considered.

## 8.2 Future Work

Some directions for future research were already discussed previously.

An important direction for future research is to extend our (space-optimal) wait-free protocol for more complex objects such as snapshot and MWMR wait-free buffers. The snapshot, which is a data structure consisting of many components simultaneously shared by many concurrent processes [1], and MWMR wait-free buffer can be constructed based on our approach to reduce their space costs.

Second, LLREF can be extended for more relaxed task arrival models including sporadic and UAM. The sporadic task model has proven useful for the modelling of recurring processes that occur in real-time systems [12], and UAM subsumes most traditional arrival models (including sporadic) as its special cases. LLREF only considers periodic task arrival and it would be an interesting research direction to overcome that restriction.

Third, it may be possible to further tighten gMUA's feasible conditions for satisfying utility lower bounds. This bound is highly related to the bound of global EDF and the effort to tighten the global EDF bound is still ongoing. More tightened bound of global EDF will further tighten the bound of gMUA.

Fourth, besides global scheduling schemes for multiprocessors that is considered in this dissertation, partitioned (UA) scheduling schemes can be considered to provide lower bounds on accrued

utility. Even though it is known that partitioned schemes cannot produce the optimal schedules in polynomial time, its reduced migration overhead is attractive for many real-time applications.

# Bibliography

- [1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. In *Journal of the ACM*, volume 40, pages 873–890, 1993.
- [2] J. Anderson. Composite registers. *Distributed Computing*, 6(3):141–154, April 1993.
- [3] J. Anderson, V. Bud, and U. C. Devi. An edf-based scheduling algorithm for multiprocessor soft real-time systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 199–208, July 2005.
- [4] J. Anderson, R. Jain, and K. Jeffay. Efficient object sharing in quantum-based real-time systems. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 346–355, 1998.
- [5] J. Anderson, R. Jain, and S. Ramamurthy. Wait-free object-sharing schemes for real-time uniprocessors and multiprocessors. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 111 – 122, Dec. 1997.
- [6] J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. *ACM Transactions on Computer Systems (TOCS)*, 15(2):134–165, 1997.
- [7] J. H. Anderson and P. Holman. Efficient pure-buffer algorithms for real-time systems. In *Proceedings of the Seventh International Conference on Real-time Computing Systems and Applications*, pages 57–64, 2000.

- [8] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 95–105, December 2001.
- [9] T. P. Baker. Stack-based scheduling of real-time processes. *Real-Time Systems*, 3(1):67–99, Mar. 1991.
- [10] T. P. Baker. Multiprocessor edf and deadline monotonic schedulability analysis. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 120–129, Dec. 2003.
- [11] S. Baruah, N. Cohen, C. G. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. In *Algorithmica*, volume 15, page 600, 1996.
- [12] S. Baruah and N. Fisher. The partitioned scheduling of sporadic real-time tasks on multiprocessor platforms. In *IEEE International Conference on Parallel Processing Workshops (ICPPW05)*, 2005.
- [13] M. Bertogna, M. Cirinei, and G. Lipari. Improved schedulability analysis of edf on multiprocessor platforms. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 209–218, 2005.
- [14] H.-J. Boehm. An almost non-blocking stack. In *ACM Symposium on Principles of Distributed Computing*, pages 40–49, 2004.
- [15] J. Carpenter, S. Funk, et al. A categorization of real-time multiprocessor scheduling problems and algorithms. In J. Y. Leung, editor, *Handbook on Scheduling Algorithms, Methods, and Models*, page 30.130.19. Chapman Hall/CRC, Boca Raton, Florida, 2004.
- [16] CCRP. Network centric warfare. <http://www.dodccrp.org/ncwPages/ncwPage.html>.
- [17] A. Chandra, M. Adler, and P. Shenoy. Deadline fair scheduling: Bridging the theory and practice of proportionate-fair scheduling in multiprocessor servers. In *Proceedings of the 21st IEEE Real-Time Technology and Applications Symposium*, pages 3–14, 2001.

- [18] J. Chen. A loop-free asynchronous data sharing mechanism in multiprocessor real-time systems based on timing properties. In *The First International Workshop on Data Distribution for Real-Time Systems (DDRTS) 2003*, pages 184–190, 2003.
- [19] J. Chen and A. Burns. A fully asynchronous reader/writer mechanism for multiprocessor real-time systems. Technical Report YCS-288, CS Dept., University of York, May 1997.
- [20] J. Chen and A. Burns. A three-slot asynchronous reader/writer mechanism for multiprocessor real-time systems. Technical Report YCS-186, Department of Computer Science, University of York, 1997.
- [21] K. Chen and P. Muhlethaler. A scheduling algorithm for tasks described by time value function. *Journal of Real-Time Systems*, 10(3):293–312, May 1996.
- [22] H. Cho, B. Ravindran, and E. D. Jensen. An optimal real-time scheduling for multiprocessors. In *IEEE Real-Time Systems Symposium (RTSS)*, under Review.
- [23] H. Cho, B. Ravindran, and E. D. Jensen. A space-optimal, wait-free real-time synchronization protocol. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 79–88, July 2005.
- [24] H. Cho, B. Ravindran, and E. D. Jensen. Lock-free synchronization for dynamic embedded real-time software. In *ACM/IEEE Design, Automation, and Test in Europe (DATE)*, pages 438–443, March 2006.
- [25] H. Cho, B. Ravindran, and E. D. Jensen. On utility accrual processor scheduling with wait-free synchronization for embedded real-time software. In *ACM Symposium on Applied Computing (SAC)*, pages 918 – 922, April 2006.
- [26] H. Cho, H. Wu, B. Ravindran, and E. D. Jensen. On multiprocessor utility accrual real-time scheduling with statistical timing assurances. In *IFIP Embedded and Ubiquitous Computing (EUC)*, Aug 2006.
- [27] R. K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, Carnegie Mellon University, 1990.

- [28] R. K. Clark, E. D. Jensen, et al. An adaptive, distributed airborne tracking system. In *IEEE International Workshop Parallel and Distributed Real-Time Systems*, pages 353–362, April 1999.
- [29] R. K. Clark, E. D. Jensen, and N. F. Rouquette. Software organization to facilitate dynamic processor scheduling. In *International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, April 2004.
- [30] M. L. Dertouzos and A. K. Mok. Multiprocessor on-line scheduling of hard real-time tasks. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, December 1989.
- [31] U. C. Devi and J. Anderson. Tardiness bounds for global edf scheduling on a multiprocessor. In *IEEE Real-Time Systems Symposium (RTSS)*, 2005.
- [32] U. C. Devi, H. Leontyev, and J. Anderson. Efficient synchronization under global edf scheduling on multiprocessors. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 75–84, July 2006.
- [33] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127140, 1978.
- [34] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo. A new kernel approach for modular real-time systems development. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 199–206, 2001.
- [35] GlobalSecurity.org. Mprtip. <http://www.globalsecurity.org/intell/systems/mp-rtip.htm/>.
- [36] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic tasks systems on multiprocessors. *Real-Time Systems*, 25(2-3):187–205, 2003.
- [37] M. G. Gouda, Y.-W. Han, E. D. Jensen, W. D. Johnson, and R. Y. Kain. Radar scheduling: Section 1, the scheduling problem. In *Distributed Data Processing Technology, Applications*

*of DDP Technology to BMD: Architectures and Algorithms*, volume IV, chapter 3. Honeywell Systems and Research Center, Minneapolis, MN, September 1977.

- [38] P. T. Hakan Sundell. Space efficient wait-free buffer sharing in multiprocessor real-time systems based on timing information. In *IEEE International Conference on Real-Time Computing Systems and Applications (RTCISA)*, pages 433–440, 2000.
- [39] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 522–529, 2003.
- [40] J.-F. Hermant and G. L. Lann. A protocol and correctness proofs for real-time high-performance broadcast networks. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 360–369, 1998.
- [41] M. Hohmuth and H. Hartig. Pragmatic nonblocking synchronization for real-time systems. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 217–230, 2001.
- [42] P. Holman and J. H. Anderson. Locking in pfair-scheduled multiprocessor systems. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 149–158, 2002.
- [43] P. Holman and J. H. Anderson. Object sharing in pfair-scheduled multiprocessor systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 111–120, 2002.
- [44] P. Holman and J. H. Anderson. Adapting pfair scheduling for symmetric multiprocessors. In *Journal of Embedded Computing*, volume 1, pages 543–564, 2005.
- [45] W. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21:177–185, 1974.
- [46] H. Huang, P. Pillai, and K. G. Shin. Improving wait-free algorithms for interprocess communication in embedded real-time systems. In *USENIX Annual Technical Conference*, pages 303–316, 2002.

- [47] E. D. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time systems. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 112–122, December 1985.
- [48] H. Kopetz and J. Reisinger. The non-blocking write protocol nbw: A solution to a real-time synchronisation problem. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 131–137, 1993.
- [49] G. Koren and D. Shasha. D-over: An optimal on-line scheduling algorithm for overloaded real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 290–299, December 1992.
- [50] P. Li. *Utility Accrual Real-Time Scheduling: Models and Algorithms*. PhD thesis, ECE Dept., Virginia Polytechnic Institute and State University, 2004. <http://scholar.lib.vt.edu/theses/available/etd-08092004-230138/>.
- [51] P. Li and B. Ravindran. Fast, best effort real-time scheduling algorithms. *IEEE Transactions on Computers*, 53(9):1159 – 1175, 2004.
- [52] P. Li, B. Ravindran, et al. A formally verified application-level framework for real-time scheduling on posix real-time operating systems. *IEEE Transactions on Software Engineering*, 30(9):613 – 629, Sept. 2004.
- [53] P. Li, B. Ravindran, and E. D. Jensen. Utility Accrual Resource Access Protocols with Assured Timeliness Behavior for Real-Time Embedded Systems. In *Proc. 14th International Conference on Real-Time and Network Systems (RTNS06)*, May 2006.
- [54] P. Li, B. Ravindran, H. Wu, and E. D. Jensen. A utility accrual scheduling algorithm for real-time activities with mutual exclusion resource constraints. *IEEE Transactions on Computers*, 55:454–469.
- [55] K. Lin, Y. Wang, T. Chien, and Y. Yeh. Designing multimedia applications on real-time systems with smp architecture. In *IEEE Fourth International Symposium on Multimedia Software Engineering*, page 17, 2002.

- [56] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, Carnegie Mellon University, 1986.
- [57] D. P. Maynard, S. E. Shipman, et al. An example real-time command, control, and battle management application for alpha. Technical report, CMU CS Dept., Dec. 1988. Archons Project TR 88121.
- [58] M. M. Michael and M. L. Scott. Non-blocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, May 1998.
- [59] A. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. PhD thesis, Department of EE and CS, MIT, 1983.
- [60] D. Mosse, M. E. Pollack, and Y. Ronen. Value-density algorithm to handle transient overloads in scheduling. In *IEEE Euromicro Conference on Real-Time Systems*, pages 278–286, June 1999.
- [61] J. D. Northcutt. *Mechanisms for Reliable Distributed Real-Time Operating Systems – The Alpha Kernel*. Academic Press, 1987.
- [62] QNX. Symmetric multiprocessing. <http://www.qnx.com/products/rtos/smp.html>. Last accessed October 2005.
- [63] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 116–123, 1990.
- [64] B. Ravindran, E. D. Jensen, and P. Li. On recent advances in time/utility function real-time scheduling and resource management. In *IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, pages 55 – 60, May 2005.
- [65] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

- [66] P. Sorensen and V. Hemacher. A real-time system design methodology. In *INFOR 13*, pages 1–18, 1975.
- [67] A. Srinivasan and J. Anderson. Efficient scheduling of soft real-time applications on multiprocessors. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 51–59, July 2003.
- [68] A. Srinivasan and S. Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. In *Information Processing Letters*, pages 93–98, Nov 2002.
- [69] A. Srinivasan, P. Holman, J. Anderson, and S. Baruah. The case for fair multiprocessor scheduling. In *IEEE International Parallel and Distributed Processing Symposium*, page 1143, 2003.
- [70] H. Sundell and P. Tsigas. Space efficient wait-free buffer sharing in multiprocessor real-time systems based on timing information. In *IEEE International Conference on Real-Time Computing Systems and Applications (RTCISA)*, pages 433 – 440, 2000.
- [71] H. Sundell, P. Tsigas, and Y. Zhang. Simple and fast wait-free snapshots for real-time systems. In *4th International Conference On Principles Of Distributed Systems (OPODIS)*, pages 91–106, 2000.
- [72] R. K. Treiber. System programming: Copying with parallelism. Technical report, IBM Almaden Research Center, April 1986. RJ 5118.
- [73] P. Tsigas and Y. Zhang. Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the 2001 ACM SIGMETRICS Int'l Conf. on Measurement and Modeling of Computer Systems*, pages 320–321, 2001.
- [74] P. Tsigas and Y. Zhang. Efficient wait-free queue algorithms for real-time synchronization. Technical report, Department of Computing Science, Chalmers University of Technology, 2002.

- [75] P. Tsigas and Y. Zhang. Integrating non-blocking synchronization in parallel applications: Performance advantages and methodologies. In *Proceedings of the Third Int'l Workshop on Software and Performance*, pages 55–67, 2002.
- [76] J. Valois. Lock-free linked list using compare-and-swap. In *Proceedings of the Fourteenth ACM Symposium on Principles of Distributed Computing*, pages 214–222, 1995.
- [77] P. M. B. Vitanyi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 233–243, 1986.
- [78] C. Wang, H. Takada, and K. Sakamura. Priority inheritance spin locks for multiprocessor real-time systems. In *International Symposium on Parallel Architectures, Algorithms, and Networks*, pages 70–76, 1996.
- [79] J. Wang and B. Ravindran. Time-utility function-driven switched ethernet: Packet scheduling algorithm, implementation, and feasibility analysis. *IEEE Transactions on Parallel and Distributed Systems*, 15(2):119–133, Feb. 2004.
- [80] H. Wu, B. Ravindran, et al. Utility accrual scheduling under arbitrary time/utility functions and multiunit resource constraints. In *IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA)*, pages 80–98, August 2004.
- [81] H. Wu, B. Ravindran, and E. D. Jensen. Energy-Efficient, Utility Accrual Real-Time Scheduling Under the Unimodal Arbitrary Arrival Model. In *ACM Design, Automation, and Test in Europe (DATE)*, March 2005.
- [82] H. Wu, B. Ravindran, E. D. Jensen, and P. Li. Cpu scheduling for statistically-assured real-time performance and improved energy-efficiency. In *IEEE/ACM/IFIP Conference on Hardware/Software Codesign and System Synthesis*, pages 110–115, September 2004.

- [83] H. Wu, B. Ravindran, E. D. Jensen, and P. Li. Energy-efficient, utility accrual scheduling under resource constraints for mobile embedded systems. In *ACM International Conference on Embedded Software*, pages 64–73, September 2004.
- [84] H. Wu, B. Ravindran, E. D. Jensen, and P. Li. Energy-Efficient, Utility Accrual Scheduling under Resource Constraints for Mobile Embedded Systems. In *ACM EMSOFT*, pages 64–73, Sept. 2004.
- [85] V. Yadaiken. Against priority inheritance. Technical report, Finite State Machine Labs, June 2002.
- [86] O. U. P. Zapata and P. M. Alvarez. Edf and rm multiprocessor scheduling algorithms: Survey and performance evaluation. <http://delta.cs.cinvestav.mx/~pmejia/multitechreport.pdf>. Last accessed October 2005.
- [87] X. Zhang, Z. Wang, et al. System support for automated profiling and optimization. In *ACM SOSP*, pages 15–26, October 1997.

# Vita

Hyeonjoong Cho is currently a Ph.D. candidate in the Bradley Department of Electrical Computer Engineering at Virginia Polytechnic Institute and State University (Virginia Tech.), USA. He received his B.S. degree in Electronic Engineering from Kyungpook National University (1996) and M.S. degree in Electronic and Electrical Engineering from Pohang University of Science and Technology, South Korea (1998). He had worked for Samsung Electronics as a senior software engineer at Factory Automation research institute before pursuing his Ph.D. degree.

His Ph.D. dissertation research focuses on real-time scheduling and synchronization for single and multiprocessors. His other research interests include real-time operating systems, embedded systems, industrial field bus, and algorithm design.